

PODSTAWY PROGRAMOWANIA W JĘZYKU PYTHON

poziom I



Cele kursu

- Nabycie praktycznych umiejętności programowania aplikacji w języku Python
- Poznanie fundamentów języka Python niezbędnych do dalszego poznawania zaawansowanych możliwości języka i technologii



Rozkład kursu

Moduł	Dzień				
	1	2	3	4	5
1. WPROWADZENIE DO JĘZYKA PYTHON	X				
2. PODSTAWOWE KONCEPCJE	X	X			
3. ZŁOŻONE TYPY DANYCH		X	X		
4. PROGRAMOWANIE FUNKCYJNE				X	
5. KLASY I OBIEKTY					X
6. MODUŁY I PAKIETY					X
7. OPERACJE NA PLIKACH					X
8. WYJĄTKI					X
9. WAŻNE WBUDOWANE MODUŁY I BIBLIOTEKI					X

Przygotowanie uczestników

- To szkolenie nie ma formalnych wymagań wstępnych
- Przydatne będą:
 - umiejętność programowania w innych językach,
np. C/C++, Java, PHP, Visual Basic
 - umiejętność logicznego, analitycznego myślenia



"By failing to prepare, you are preparing to fail."

Benjamin Franklin

Przedstawienie uczestników



Dzień dobry,
Mam na imię ...
W pracy zajmuję się ...
Mam doświadczenie w ...
Od szkolenia oczekuję ...



Plan szkolenia

- 1 WPROWADZENIE DO JĘZYKA PYTHON**
- 2 PODSTAWOWE KONCEPCJE
- 3 ZŁOŻONE TYPY DANYCH
- 4 PROGRAMOWANIE FUNKCYJNE
- 5 KLASY I OBIEKTY
- 6 MODUŁY I PAKIETY
- 7 OPERACJE NA PLIKACH
- 8 WYJĄTKI
- 9 WAŻNE WBUDOWANE MODUŁY I BIBLIOTEKI

Plan modułu

1 WPROWADZENIE DO JĘZYKA PYTHON

- czym jest Python?
- krótka historia języka
- filozofia języka (the Zen of Python)
- pierwszy program
- instalacja środowiska
- praca w trybie interaktywnym
- wybór środowiska zintegrowanego (IDE)



Python

- Python – język programowania:
 - wysokiego poziomu
 - ogólnego przeznaczenia (szerokiego zastosowania)
 - o rozbudowanym pakiecie bibliotek standardowych
- Python jest językiem skryptowym, aczkolwiek wykorzystuje się go także do dużych projektów
- Oficjalna  strona WWW

Python – cechy

- Dynamiczne typowanie
- Automatyczne zarządzanie pamięcią
- *Garbage collector*
- Zorientowany obiektowo (*Object-Oriented*)
- Wbudowane typy obiektów
- Wbudowane narzędzia
- *Batteries included*
- Open source
- Przenośny
- Łatwy w nauce
- Szybki proces implementacji

Python – zastosowanie

- Systemowe
- GUI
- Skrypty internetowe
- Aplikacje webowe
- Integracja komponentów
- Obsługa baz danych
- Błyskawiczne prototypowanie
- Matematyka i nauka
- Gry, grafika
- Eksperymentowanie z językiem

Historia Pythona

- Twórcą Pythona jest holenderski programista **Guido van Rossum**
- Python wywodzi się z języka *ABC* (pochodnej języka *Simula*)
- Pierwsza wersja interpretera Pythona powstała pod koniec roku 1989
 - 1991 r. – v0.9, pierwsza publikacja
 - 1994 r. – v1.0, elementy języka funkcyjnego
 - 2000 r. – v2.0, pełny GC, Unicode
 - 2008 r. – v3.0, brak wstecznej kompatybilności



Historia Pythona

- Choć w logo Pythona występuje wąż...



...to nazwa nawiązuje do
Latającego Cyrku Monty Pythona
– wiele odniesień do skeczów
można znaleźć w przykładach
kodu i nazwach bibliotek



Projekt Pythona

- Python jest rozwijany jako projekt  open-source
- Jest zarządzany przez  Python Software Foundation, która jest organizacją *non-profit* (na wzór *Apache Software Foundation*)

Rozwój języka

- Rozwój języka jest prowadzony przy wykorzystaniu PEP (*Python Enhancement Proposal*)
- Są to propozycje zmian lub rozszerzeń w języku w postaci krótkiego artykułu, który jest następnie poddawany dyskusji wśród programistów Pythona
- PEP zawiera:
 - opis proponowanego rozwiązania
 - uzasadnienie (*Rationale*)
 - aktualny status
- Po osiągnięciu konsensusu, propozycje są przyjmowane lub odrzucane

Filozofia Pythona

- Przewodnią ideą Pythona jest **czytelność i przejrzystość kodu**
- Składnia języka skłania programistów do tworzenia zwięzłego, eleganckiego kodu
- Podstawę “filozofii Pythona” (*The Zen of Python*) stanowi 19 aforzyzmów stworzonych przez dewelopera Pythona Tima Petersa
- Zostały one zebrane w dokumencie PEP-20
- Zasady te są dostępne w każdym interpreterze Pythona (jako tzw. *Easter Egg*) po wydaniu polecenia:

Easter Egg

```
>>> import this
```

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one - and preferably only one - obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than right now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea - let's do more of those!

Przykład aplikacji "Hello world"

x86 NASM

```
; NASM Intel 8086 Assembler (DOS): "Hello, world!"  
  
.org 100h  
  
start:  
    MOV AH, 09h  
    LEA DX, [msg]  
    INT 21h  
    MOV AX, 4C00h  
    INT 21h  
  
msg: DB 'Hello, world!', 0Dh, 0Ah, '$'
```

Przykład aplikacji "Hello world"

ANSI C

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    // printf() displays the string inside quotation
    printf("Hello, world!\n");
    return 0;
}
```

Java

```
public class HelloWorld {

    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

Przykład aplikacji "Hello world"

Perl

```
print "Hello, world!\n";
```

Python 2.x

```
print "Hello, world!"
```

Python 3.x

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

Instalacja środowiska

- Aktualna i starsze wersje interpretera Pythona (dla różnych wersji systemów operacyjnych) są dostępne do pobrania i zainstalowania ze strony  <https://www.python.org/downloads/>
- W trakcie instalacji można wskazać katalog instalacyjny oraz dodatki (*features*), jakie mają być doinstalowane
- Po instalacji można upewnić się, czy do zmiennej *PATH* został dodany katalog instalacyjny

Ćwiczenia/przykłady

- Ćwiczenie/przykład 1.1:
Instalacja środowiska z konsolą interaktywną



Praca z interpreterem

- Aby uruchomić interpreter Pythona **w trybie interaktywnym** należy w wierszu linii poleceń wpisać polecenie:

Uruchomienie interpretera w trybie interaktywnym

```
C:\Users\student> python
Python 3.10.2 (tags/v3.10.2:a58ebcc, Jan 17 2022, 14:12:15)
          [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more
information.

>>>
```

- Polecenie zawiera zarówno interpreter, jak i kompilator
- Kompilator jest uruchamiany automatycznie, tylko, gdy jest to niezbędne (np. w przypadku importowanych modułów)

Praca z interpreterem

- Od tego momentu interpreter działa w trybie powłoki – po znaku zachęty (*prompt*) `>>>` można wpisywać kolejne polecenia, które są interpretowane i wykonywane
- Aby wyjść z tego trybu należy w systemie Windows wcisnąć klawisze `<Ctrl/Z>` (`<Ctrl/D>` na systemie Unix) lub wydać polecenie `quit()`

Praca w trybie wsadowym

- Możliwa jest także praca **w trybie wsadowym** – kolejne polecenia można zapisać do pliku, a następnie go wykonać
- Typowe rozszerzenia plików Pythona:

.py	standardowy skrypt źródłowy
.pyc	skrypt skompilowany do kodu bajtowego
.pyo	kod zoptymalizowany (od Pythona 3.5 są używane tylko pliki .pyc)
.pyd	plik biblioteki z kodem Pythona spakowany jak DLL (specyficzny dla systemu Windows)
.pyw	skrypt GUI (uruchamiany za pomocą <i>pythonw.exe</i> – nie powoduje uruchomienia konsoli)

Uruchamianie skryptów – Windows

- W systemie Windows do uruchomienia skryptu Pythona o nazwie *program.py* wystarczy wydać w konsoli polecenie:

Uruchomienie skryptu

```
C:\Users\student> python program.py
```

lub jeszcze prościej:

Uruchomienie skryptu

```
C:\Users\student> program.py
```

- W tym przypadku Windows wykorzystuje swój rejestr do rozpoznania rozszerzenia i automatycznie uruchamia interpreter

Uruchamianie skryptów – Unix

- W przypadku uruchamiania skryptów na systemie Unix, uzupełnia się program o **dodatkową linię** identyfikującą położenie interpretera (tzw. *shebang line*)
- Taka informacja musi znaleźć się w pierwszej linii skryptu
- Ścieżkę do interpretera podaje się za znakami **#!** – jest to informacja wykorzystywana przez system operacyjny

Shebang line

```
#!/usr/bin/python
```

- Jeśli taki skrypt poda się do wykonania uruchomionemu interpreterowi, to linia traktowana jest jak komentarz

Uruchamianie skryptów – Unix

- Bardziej uniwersalnym podejściem jest zdefiniowanie linii *shebang* następująco:

Shebang line

```
#!/usr/bin/env python
```

- W tym przypadku ścieżka do interpretera będzie poszukiwana w bieżącym środowisku powłoki

Uruchamianie skryptów – Unix

- Należy także pamiętać o nadaniu skryptowi uprawnienia do wykonania:

Nadanie uprawnień

```
$ chmod +x program.py  
$ ./program.py
```

Uruchamianie skryptów – Windows

- Do uruchomienia skryptu w systemie Windows można użyć polecenia:

Uruchomienie skryptu

```
C:\Users\student> py program.py
```

- Program **py.exe** to tzw. *Windows launcher* (PEP-397) – uruchomi skrypt zgodnie ze wskazaną wersją Pythona w linii *shebang*
- Można także wskazać wersję Pythona poprzez argument polecenia

Powłoka interpretera

- **Powłoka interpretera (*shell*)** – doskonały dodatek, którego brakuje w wielu innych językach programowania
- Wiele ciekawych nakładek:
 -  [IPython](#)
 -  [bpython](#)
 -  [ptpython](#)
- Możliwości IPython'a:
 - bufor klawiatury
 - kolorowanie składni
 - skróty klawiaturowe
 - dopełnienie nazw
 - edycja multilinii

Środowisko pracy

- IDE (*Integrated Development Environment*) – zintegrowane środowisko programistyczne to aplikacja lub zespół aplikacji (środowisko) projektowany z myślą o maksymalizacji produktywności programisty
- Obecnie na rynku dostępnych jest wiele **zintegrowanych środowisk programistycznych** dedykowanych dla programistów określonego języka lub o charakterze uniwersalnym
- Często posiadają bezpłatne, okrojone wersje

Typowe cechy środowiska pracy

- Bogaty edytor kodu – uzupełnianie składni, oznaczanie błędów i ostrzeżeń, podpowiadanie sposobów usunięcia błędów, refactoring, automaty do generowania kodu
- Automatyczne budowanie projektu
- Możliwość szybkiego uruchomienia projektu
- Szybkie przemieszczanie się po projekcie
- Tworzenie dokumentacji za pomocą prostych kreatorów
- Możliwość uruchomienia aplikacji w trybie debugowania
- Profilowanie aplikacji
- . . . i wiele innych, w zależności od produktu

Eclipse

- Do Eclipse'a można doinstalować wtyczkę dla Pythona
- W tym celu należy:
 - z menu wybrać polecenie: *Help → Install new software...*
 - wskazać lokalizację <http://pydev.org/updates>, skąd można pobrać wtyczkę
 - wybrać z listy kategorię *PyDev*
 - rozpocząć instalację

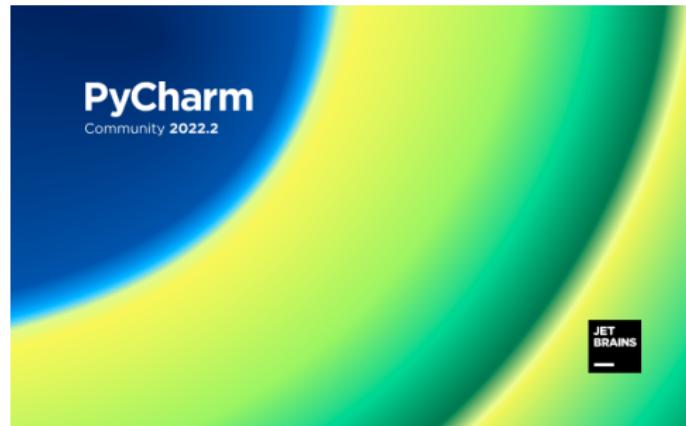


- Do *IntelliJ IDEA* można także doinstalować wtyczkę dla Pythona
- Standardowo nie jest ona dołączana
- Należy:
 - wybrać opcję: *Configure → Settings*
 - wybrać kategorię *Plugins*
 - wcisnąć przycisk *Install JetBrains plugin...*
 - z listy dostępnych wtyczek wybrać *Python Community Edition*
 - wcisnąć przycisk *Install*
- Po zakończeniu instalacji środowisko IDE wymaga zrestartowania



PyCharm IDE

- Można też wykorzystać komercyjne środowiska IDE dedykowane dla Pythona, np. takie jak:  **PyCharm** firmy *JetBrains*



- Aktualnie dostępna jest także okrojona, darmowa wersja *Community* – można ją pobrać  *stąd*

Ćwiczenia/przykłady

- Ćwiczenie/przykład 1.2:
Instalacja IDE



ĆWICZENIA

Plan szkolenia

1 WPROWADZENIE DO JĘZYKA PYTHON

2 PODSTAWOWE KONCEPCJE

3 ZŁOŻONE TYPY DANYCH

4 PROGRAMOWANIE FUNKCYJNE

5 KLASY I OBIEKTY

6 MODUŁY I PAKIETY

7 OPERACJE NA PLIKACH

8 WYJĄTKI

9 WAŻNE WBUDOWANE MODUŁY I BIBLIOTEKI

Plan modułu

2 PODSTAWOWE KONCEPCJE

- identyfikatory
- bloki danych
- komentarze
- zmienne
- instrukcje podstawienia
- typy wbudowane (proste) i operatory
- instrukcje sterujące – instrukcje warunkowe
- instrukcje sterujące – instrukcje powtarzania (pętle)



Identyfikatory

- **Identyfikatory** (*identifiers*) – nazwy nadawane zmiennym, funkcjom, klasom, modułom
- Prawidłowy identyfikator składa się ze:

znaku początkowego | dowolnej litery z tabeli Unicode (w tym także znaków narodowych i znaku podkreślenia _)

znaków kontynuacji | dowolnej ilości znaków początkowych oraz cyfr z tabeli Unicode

- W identyfikatorach istotna jest wielkość liter (*case-sensitive*)

Identyfikatory – dobre praktyki

- Idenfikator nie może być identyczny z żadnym ze **słów kluczowych języka**
- Należy unikać używania jako własnych identyfikatorów **nazw predefiniowanych identyfikatorów Pythona** (nazw wbudowanych typów danych, funkcji, wyjątków)
- Nie należy jako własnych identyfikatorów stosować nazw, które **rozpoczynają się i kończą dwoma podkreśleniami** – Python używa takich identyfikatorów jako **nazw specjalnych atrybutów, metod i zmiennych**, specyficznych dla języka

Identyfikatory

- Zgodnie z przedstawionymi regułami pojedynczy znak podkreślenia _ jest także **poprawnym identyfikatorem**
- Jest często wykorzystywany przy internacjonalizacji aplikacji
- Można go także wykorzystać w celu zignorowania specyficznych wartości
- Znak podkreślenia _ w trybie interaktywnym reprezentuje **wynik ostatniego wyrażenia**

Identyfikatory – konwencje nazewnicze

- **Nazwy klas** – rozpoczynamy dużą literą i stosujemy konwencję *CamelCase*
- **Pozostałe identyfikatory** – rozpoczynamy małą literą i stosujemy konwencję *snake_case*

Linie i wcięcia

- **Blok kodu (suite)** jest identyfikowany **na podstawie wcięć** – w Pythonie nie ma nawiasów, ani słów kluczowych wskazujących początek i koniec bloku
- Głębokość wcięcia nie jest ustalona – ogólnie przyjętym standardem jest głębokość 4 spacji na poziom
- Ciągły zbiór linii z tym samym wcięciem formuje blok kodu
- Znak dwukropka jest wykorzystywany do powiązania bloku kodu z instrukcją sterującą (instrukcją warunkową, pętlą, itp.)

Instrukcje i wyrażenia wielowierszowe

- Instrukcje i wyrażenia w Pythonie kończą się wraz z końcem linii
- Jeżeli w linii występuje tylko jedna instrukcja, to nie ma potrzeby kończenia jej średnikiem (jak w wielu innych, popularnych językach)
- Jeżeli instrukcja ma być kontynuowana w kolejnej linii, to poprzednią należy zakończyć **znakiem kontynuacji** \
- Instrukcje wykorzystujące nawiasy (), [] oraz {} mogą obejmować wiele linii i **nie wymagają użycia znaku kontynuacji**

Wiele instrukcji w jednej linii

- W jednej linii można umieścić wiele instrukcji
- Wtedy trzeba je oddzielić znakiem **średnika**

Przykład

```
>>> a = 5; b = 7; c = a + b
>>> print(c)
12
>>>
```

- Do wypisania wartości zmiennej wykorzystuje się funkcję *print*
- W trybie interaktywnym powłoki można pominąć wywołanie funkcji i wpisać tylko nazwę zmiennej lub wyrażenie

Wypisywanie wartości

- Za pomocą funkcji *print* można jednocześnie wypisać wiele wartości
- Standardowo są one separowane pojedynczą spacją – można to zmienić za pomocą parametru *sep*
- Standardowo po wypisaniu wartości za pomocą funkcji *print* następuje przejście do nowej linii – można to zmienić za pomocą parametru *end*

Wczytywanie wartości

- Do wprowadzenia danych z klawiatury do programu służy funkcja `input`
- Poprzez argument można podać opcjonalny tekst podpowiedzi – dzięki temu użytkownik będzie wiedział, że program oczekuje wprowadzenia danych
- Wczytane przez funkcję dane mają **postać tekstową**

Wprowadzanie danych

```
>>> name = input('Podaj imię: '); print('Witaj', name)
Podaj imię: John
Witaj John
```

Komentarze

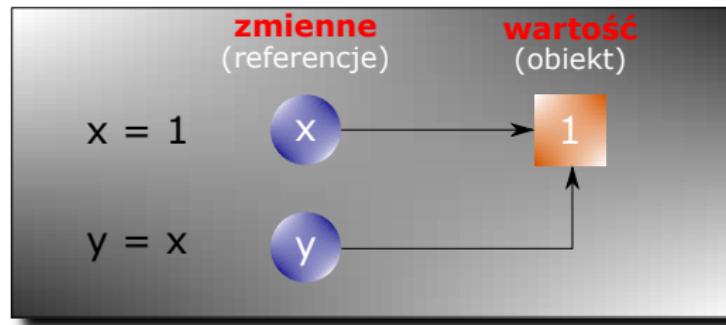
- **Komentarze** w Pythonie umieszczane są za znakiem **#** (o ile nie znajduje się on w literale tekstowym)
- Tekst komentarza znajdujący się **za znakiem # do końca linii** jest ignorowany przez interpreter
- Podobnie interpreter ignoruje linie zawierające wyłącznie znaki białe
- W ogólności ten rodzaj komentarzy służy do opisu kodu **z punktu widzenia dewelopera** – ma ułatwić zrozumienie, co kod robi

Komentarze dokumentujące

- Dłuższe komentarze, obejmujące wiele linii, można umieścić w kodzie stosując literały tekstowe ograniczone **potrójnymi apostrofami**, bądź **potrójnymi cudzysłowami**
- Standardowo umieszcza się je na początku modułu, funkcji, klasy, metody
- Tego typu komentarze (tzw. *docstrings – documentation strings*) wykorzystuje się do tworzenia dokumentacji
- Mają na celu wyjaśnić, jaką rolę pełni opisany kod i jak go użyć, **z punktu widzenia użytkownika**
- Do automatycznego utworzenia dokumentacji wykorzystuje się moduł *pydoc*

Zmienne

- W Pythonie wszystkie dane są **obiektami** (*objects*)
- **Zmienna** (*variable*) – nazwa referencji (odniesienia) do zarezerwowanego obszaru pamięci, w którym przechowywana jest dana (wartość)



Mutowalność danych

- **Obiekt zmienny** (*mutable object*) to obiekt modyfikowalny – operacje wykonywane na takim obiekcie mogą zmienić jego stan (zmianie podlegają wartości danych, ale nie ich lokalizacja w pamięci) – zmiany są dokonywane **w miejscu**
- Operacje na obiektach mutowalnych nie zwracają żadnego konkretnego obiektu, lecz wartość **None**
- **Obiekt niezmienny** (*immutable object*) – każda operacja modyfikująca stan obiektu powoduje utworzenie **nowego obiektu** (oryginalny obiekt pozostaje “**nietknięty**”)

Typy danych

- Każda dana w Pythonie posiada swój **typ**
- Typ decyduje o:
 - operacjach, jakie można wykonać na obiekcie
 - posiadanych atrybutach
- Typ danej nie może samoczynnie (np. w zależności od kontekstu) ulec zmianie – każda zmiana wymaga **jawniej konwersji** typu

Instrukcje podstawienia

- Do **nadania wartości zmiennej** służy operator przypisania `=` (*assignment operator*)
- Taka operacja **nie wymaga wcześniejszej deklaracji** zmiennej
- Typ zmiennej jest **wnioskowany** na podstawie przypisanej wartości

Przykład instrukcji podstawienia

```
>>> integer_value = 123      # zmienna całkowita  
>>> decimal_value = 3.21     # zmienna zmiennoprzecinkowa  
>>> string_value = 'Python'  # zmienna tekstowa (łańcuch znakowy)
```

Instrukcje podstawienia

- Ta sama zmienna może być użyta do wskazania innej wartości (niekoniecznie tego samego typu)
- Do sprawdzenia bieżącego typu wartości wskazywanej przez zmienną można użyć funkcji *type*

Przykład typowania dynamicznego

```
>>> zmienna = 1
>>> print(type(zmienna))
<class 'int'>

>>> zmienna = 1.0
>>> print(type(zmienna))
<class 'float'>
```

Instrukcje podstawienia

- Przypisanie jednej wartości do wielu zmiennych:

Przykład

```
>>> x = y = z = 1234
```

Można też tak...

```
>>> z = 1234
```

```
>>> y = z
```

```
>>> x = y
```

Ćwiczenia/przykłady

- Ćwiczenie/przykład 2.1:
Praca z dokumentacją – kostka do gry



ĆWICZENIA

Typy numeryczne

- W Pythonie **typy numeryczne** służą do przechowywania wartości liczbowych
- Python wspiera 3 odrębne typy numeryczne reprezentujące:
 - liczby całkowite (*integral types*), w tym wartości logiczne (*booleans*)
 - liczby zmiennoprzecinkowe (*floating point numbers*)
 - liczby zespolone (*complex numbers*)
- Wszystkie powyższe typy są **niemutowalne**

Typ całkowity

- **Typ całkowity – *int* –**

reprezentuje liczby całkowite ze znakiem, o dowolnej, **nieograniczonej precyzyji**
(od Pythona 3.x)



- Można je podawać w różnych systemach pozycyjnych: dziesiętnie, binarnie, ósemkowo, heksadecymalnie
- Od Pythona 3.6 można stosować w literałach liczbowych znak podkreślenia, jako separator cyfr

Typ całkowity – przykłady

Przykłady literałów całkowitych

```
>>> # literały całkowite dziesiętne
... i1 = 30; i2 = -1_000
>>> i1, i2
(30, -1000)

>>> # literały całkowite binarne - przedrostek 0b, 0B
... i3 = 0b111; i4 = 0B100
>>> i3, i4
(7, 4)

>>> # literały całkowite ósemkowe - przedrostek 0o, 0O
>>> #           w Pythonie 2.x - przedrostek 0
... i5 = 0o15; i6 = 0O74
>>> i5, i6
(13, 60)

>>> # literały całkowite szesnastkowe - przedrostek 0x, 0X
... i7 = 0x19; i8 = 0XAf
>>> i7, i8
(25, 175)
```

- Do konwersji wartości innych typów na typ całkowity można posłużyć się funkcją `int`

Operacje arytmetyczne

- Poniższe operatory można zastosować do danych numerycznych dowolnego typu (liczby całkowite, zmienne przecinkowe i zespolone)

OPERATOR	SPOSÓB UŻYCIA	DZIAŁANIE
+	+a	tożsamość (operator 1-arg.)
	a + b	dodawanie (operator 2-arg.)
-	-a	negacja (operator 1-arg.)
	a - b	odejmowanie (operator 2-arg.)
*	a * b	mnożenie
/	a / b	dzielenie zmienne przecinkowe
**	a ** b	potęgowanie

Operacje arytmetyczne

- W stosunku do liczb całkowitych i zmiennoprzecinkowych można zastosować dodatkowo operatory:

OPERATOR	SPOSÓB UŻYCIA	DZIAŁANIE
//	a // b	dzielenie całkowite (<i>floor division</i>)
%	a % b	modulo (reszta z dzielenia całkowitego)

- Podczas dzielenia całkowitego iloraz jest zaokrąglany do najbliższej wartości całkowitej **w dół** (tzn. w stronę ujemnej nieskończoności)
- Operacje, w których z prawej strony operatora /, // lub % będzie zero zakończą się **niepowodzeniem** (wystąpi wyjątek)

Wbudowane funkcje numeryczne

FUNKCJA	DZIAŁANIE
abs(x)	zwraca wartość bezwzględną x
divmod(x, y)	zwraca krotkę dwóch wartości całkowitych $(x // y, x \% y)$
pow(x, y) pow(x, y, z)	odpowiednik operatora $**$ (zwraca wartość x^y) szybsza alternatywa dla wyrażenia $(x ** y) \% z$
round(x) round(x, n)	zaokrąglą liczbę x do liczby całkowitej (wynik jest typu <i>int</i>) gdy n jest liczbą ujemną – zaokrąglą liczbę x do n cyfr w części całkowitej gdy n jest liczbą dodatnią – zaokrąglą liczbę x do n cyfr w części ułamkowej
bin(x)	zwraca w postaci tekstu bitową reprezentację liczby x
hex(x)	zwraca w postaci tekstu szesnastkową reprezentację liczby x
oct(x)	zwraca w postaci tekstu ósemkową reprezentację liczby x

Operacje przypisania

- **Operacje przypisania** (*assignment operations*) służą do nadania wartości zmiennej po lewej stronie operatora

OPERATOR	SPOSÓB UŻYCIA	DZIAŁANIE
=	a = value	przypisuje wartość zmiennej
+=	a += value	równoważne: a = a + value
-=	a -= value	równoważne: a = a - value
*=	a *= value	równoważne: a = a * value
/=	a /= value	równoważne: a = a / value
//=	a // value	równoważne: a = a // value
%=	a %= value	równoważne: a = a % value
**=	a **= value	równoważne: a = a ** value

Operacje bitowe

- **Operacje bitowe** (*bitwise operations*) można wykonywać na odpowiadających sobie bitach liczb całkowitych:

OPERATOR	SPOSÓB UŻYCIA	DZIAŁANIE
	a b	bitowa alternatywa – suma bitowa, bitowe "lub" (<i>bitwise or</i>)
&	a & b	bitowa koniunkcja – iloczyn bitowy, bitowe "i" (<i>bitwise and</i>)
^	a ^ b	bitowa alternatywa wykluczająca – bitowe "albo" (<i>bitwise exclusive or</i>)
<<	a << n	przesunięcie bitów liczby a o n pozycji w lewo
>>	a >> n	przesunięcie bitów liczby a o n pozycji w prawo
~a	~a	odwrócenie bitów

Typ logiczny

- **Typ logiczny** – *bool* – reprezentuje wartości logiczne
- Jest podtypem typu *int*
- Posiada dwie wartości:
 - prawdzie – odpowiada wartość **True**
 - fałszowi – wartość **False**



Kontekst logiczny

- Dla dowolnego obiektu można określić **kontekst logiczny**
- Tak, **jak wartość `False`** zachowują się:
 - zerowa wartość dowolnego typu numerycznego
 - wartość `None`
 - dowolny pusty kontener (zmienna tekstowa, lista, krotka, zbiór, słownik)
 - obiekty klas, które posiadają zdefiniowaną:
 - metodę `__bool__` zwracającą `False`
 - metodę `__len__` zwracającą zero
- **Wszystkie pozostałe** wartości są interpretowane **jak wartość `True`**
- Do sprawdzenia kontekstu logicznego lub jawnej konwersji na typ logiczny można użyć funkcji `bool`

Operacje logiczne

OPERATOR	SPOSÓB UŻYCIA	DZIAŁANIE
not	<code>not a</code>	operator negacji – zwraca wartość <i>True</i> , gdy argument <i>a</i> ma kontekst fałszywy i na odwrót
or	<code>a or b</code>	jeśli argument <i>a</i> ma kontekst prawdziwy, to operator zwraca <i>a</i> , w przeciwnym razie <i>b</i>
and	<code>a and b</code>	jeśli argument <i>a</i> ma kontekst fałszywy, to operator zwraca <i>a</i> , w przeciwnym razie <i>b</i>

- Operatory **or** i **and** są **skracające** (*short-circuit*) – jeśli wynik operacji da się ustalić na podstawie wartości pierwszego argumentu, to drugi nie będzie ewaluowany

Operacje logiczne – przykłady

Przykłady operacji logicznych

```
>>> a = 7
>>> print(not a == 4)    # operator not ma niższy priorytet niż ==
True
>>> print(not a)          # argument nie musi być typu logicznego
False

>>> print(0 or 2)
2
>>> print(1 or 2)
1
>>> print(0 and 2)
0
>>> print(1 and 2)
2
```

Operacje logiczne

- Operator skracający **or** można wykorzystać praktycznie, np. do zainicjowania “pustej” zmiennej – wartością domyślną

Wykorzystanie operatora *or*

```
>>> imie = 'Artur'  
>>> imie = imie or 'Jan'  
>>> print(imie)  
Artur
```

```
>>> imie = ''  
>>> imie = imie or 'Jan'  
>>> print(imie)  
Jan
```

Ćwiczenia/przykłady

- Ćwiczenie/przykład 2.2:
Lata przestępne



ĆWICZENIA

Typ zmiennoprzecinkowy

- **Typ zmiennoprzecinkowy – *float*** – reprezentuje liczby zmiennoprzecinkowe, tzn. posiadające część ułamkową (jest implementowany zwykle za pomocą typu podwójnej precyzji standardu IEEE-754)



Typ zmiennoprzecinkowy – przykłady

Przykłady literałów zmiennoprzecinkowych

```
>>> print(1.234_567)
```

```
1.234567
```

```
>>> print(.12)
```

```
0.12
```

```
>>> print(1.)
```

```
1.0
```

```
>>> print(2e-2)
```

```
0.02
```

```
>>> print(1.234E+2)
```

```
123.4
```

- Do konwersji na typ zmiennoprzecinkowy można użyć funkcji *float*

Typ zmiennoprzecinkowy

- Zakres wartości, precyzyje, sposób reprezentacji typu dla danej maszyny określa struktura `sys.float_info`

Stałe struktury `sys.float_info`

```
>>> from sys import float_info  
  
>>> print(float_info.max)  
1.7976931348623157e+308  
  
>>> print(float_info.min)  
2.2250738585072014e-308  
  
>>> print(float_info.epsilon)  
2.220446049250313e-16
```

Operacje porównania

- **Operator relacyjne**, inaczej **operatory porównania** (*comparison operators*) porównują dwie wartości i zwracają wynik porównania – wartość logiczną

OPERATOR	SPOSÓB UŻYCIA	OPERATOR SPRAWDZA, CZY . .
<code>==</code>	<code>a == b</code>	oba argumenty mają tę samą wartość
<code>!=</code>	<code>a != b</code>	oba argumenty mają różne wartości
<code>></code>	<code>a > b</code>	pierwszy argument jest większy od drugiego
<code>>=</code>	<code>a >= b</code>	pierwszy argument jest równy lub większy od drugiego
<code><</code>	<code>a < b</code>	pierwszy argument jest mniejszy od drugiego
<code><=</code>	<code>a <= b</code>	pierwszy argument jest równy lub mniejszy od drugiego

Operacje porównań

- Operacje porównań można łączyć w łańcuchy (*comparison chaining*)
- Wyniki cząstkowych porównań są scalane w wynik końcowy za pomocą operatora **and**
- Tu również działa skracanie – jeśli wynik wcześniejszego porównania będzie fałszywy, to dalsze operandy nie będą ewaluowane

Operacje porównań – przykład

Przykład porównań w łańcuchu

```
>>> a = 1; b = 4; c = 4; d = 6
>>> print(a < b <= c < d)
True
>>> # jest równoważne:
... print(a < b and b <= c and c < d)
True
```

Uwaga:

- w powyższym przykładzie w łańcuchu wszystkie wyrażenia są ewaluowane nie więcej niż jeden raz!
- łańcuch porównań definiuje relacje tylko pomiędzy sąsiednimi argumentami – w ogólności nie można wnioskować żadnych zależności pomiędzy wartościami niesąsiadującymi, np.: z łańcucha porównań: $a \neq b \neq c$ (czyli z $a \neq b$ and $b \neq c$) najmniej nie wynika, że: $a \neq c$

Operatory tożsamości

- Operatory tożsamości (*identity operators*) porównują **lokalizacje obiektów w pamięci** (równość referencji, równość zmiennych)

OPERATOR	SPOSÓB UŻYCIA	DZIAŁANIE
is	$a \text{ is } b$	zmienne są identyczne (wskazują ten sam obiekt)
is not	$a \text{ is not } b$	zmienne nie są identyczne (wskazują różne obiekty)

- Operator `==` porównuje **wartości** obiektów (danych)

Przykłady porównań wartości i badania tożsamości

```
>>> x = 1
>>> y = 1.0
>>> print(x == y, x is y)
True False
```

Typ zmiennoprzecinkowy

- Standardowa biblioteka definiuje dodatkowo typy zawarte w modułach:

fractions typ *Fraction* reprezentujący **liczby wymierne**

decimal typ *Decimal* reprezentujący **liczby zmiennoprzecinkowe o precyzji zdefiniowanej przez użytkownika**

Klasa *Fraction* – przykłady

Przykłady liczb wymiernych

```
>>> from fractions import Fraction
```

```
>>> print(Fraction(-12, 32))  
-3/8
```

```
>>> print(Fraction(7))  
7
```

```
>>> print(Fraction())  
0
```

```
>>> print(Fraction('-0.4'))  
-2/5
```

```
>>> print(Fraction('125e-2'))  
5/4
```

Klasa *Decimal* – przykłady

Przykłady liczb dziesiętnych

```
>>> from decimal import *
>>> print(Decimal(1)/Decimal(3))
0.333333333333333333333333333333333
>>> getcontext().prec = 2
>>> print(Decimal(1)/Decimal(3))
0.33
>>> getcontext().prec = 5
>>> print(Decimal(1)/Decimal(3))
0.33333
```

Funkcje matematyczne

- Funkcje dedykowane do wykonywania operacji na liczbach zmiennoprzecinkowych są zebrane w bibliotece matematycznej (moduł *math*)

FUNKCJA	DZIAŁANIE
math.ceil(x)	najmniejsza liczba całkowita większa lub równa x
math.floor(x)	największa liczba całkowita mniejsza lub równa x
math.trunc(x)	część całkowita liczby x (podobnie jak int(x))
math.copysign(x, y)	zwraca wartość x ze znakiem identycznym jak y
math.degrees(r)	zamienia radiany na stopnie
math.radians(d)	zamienia stopnie na radiany
math.fabs(x)	$ x $ - wartość bezwzględna x
math.factorial(x)	$x!$ - silnia x
math.fmod(x, y)	funkcja modulo
math.frexp(x)	krotka (m, e) zawierająca mantysę m i cechę e , tzn.: $x = m * 2^e$
math.ldexp(m, e)	zwraca wartość $m * 2^e$
math.fsum(i)	suma elementów iterowalnych i
math.hypot(x, y)	długość przeciwpłaszczyznowej $\sqrt{x^2 + y^2}$

Wybrane funkcje matematyczne

FUNKCJA	DZIAŁANIE
<code>math.exp(x)</code>	zwraca e^x
<code>math.expm1(x)</code>	zwraca $e^x - 1$
<code>math.log(x[, base])</code>	logarytm $\log_{base}x$ domyslna wartością podstawy jest <code>math.e</code>
<code>math.log2(x)</code>	logarytm \log_2x
<code>math.log10(x)</code>	logarytm dziesiętny $\log_{10}x$
<code>math.log1p(x)</code>	logarytm naturalny $\ln(1 + x)$
<code>math.modf(x)</code>	zwraca krotkę zawierającą część ułamkową i całkowitą x
<code>math.pow(x, y)</code>	potęgowanie x^y

- Dodatkowo biblioteka `math` zawiera:
 - funkcje trygonometryczne
 - funkcje hiperboliczne
 - funkcje specjalne
 - oraz stałe: `math.e`, `math.pi`, `math.tau` (2π), `math.inf`, `math.nan`

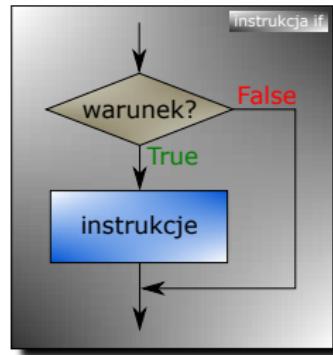
Ćwiczenia/przykłady

- Ćwiczenie/przykład 2.3:
Stan lokaty
- Ćwiczenie/przykład 2.4:
Losowanie lotto



Instrukcja warunkowa *if*

- **Instrukcja warunkowa** umożliwia wykonanie bloku kodu, tylko wtedy, gdy jest spełniony podany warunek testu → **warunkowe wykonanie bloku kodu**

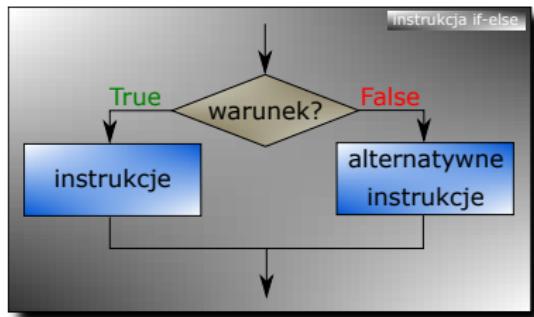


Składnia instrukcji warunkowej *if*

```
if expression:  
    statement(s)
```

Instrukcja warunkowa *if-else*

- Instrukcję warunkową można uzupełnić o **opcjonalną** sekcję **else**, która jest wykonywana, gdy warunek testowy będzie fałszywy



Składnia instrukcji warunkowej *if-else*

```
if expression:  
    statement(s)      # wykona się, jeśli warunek będzie prawdziwy  
else:  
    statement(s)      # wykona się, jeśli warunek będzie fałszywy
```

Instrukcja warunkowa *if-else* – przykład

Przykład użycia instrukcji warunkowej

```
>>> wiek = int(input('Podaj swój wiek: '))
Podaj swój wiek: 20
>>> if wiek < 0 or wiek > 150:
...     print('Nie możesz mieć tylu lat')
... else:
...     print('Masz', wiek, 'lat')
...
Masz 20 lat
```

Wyrażenie warunkowe

- W niektórych sytuacjach instrukcję warunkową *if-else* da się uprościć zastępując ją **wyrażeniem warunkowym** (*conditional expression*)

Składnia wyrażenia warunkowego

```
expression1 if boolean_expression else expression2
```

- Jest to odpowiednik **operatora trójargumentowego** ?: (*ternary operator*), znanego z języków takich jak C, czy Java

Wyrażenie warunkowe – przykład

- Instrukcję warunkową:

Instrukcja warunkowa

```
if liczba_punktow > 60:  
    egzamin = "zaliczony"  
else:  
    egzamin = "niezaliczony"
```

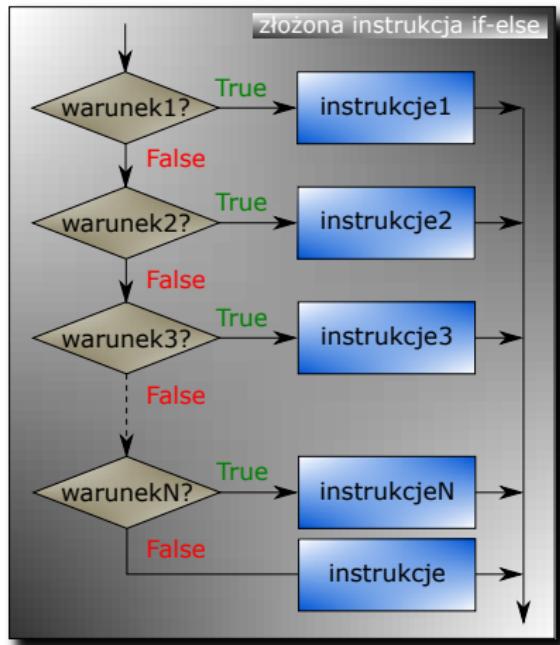
da się zapisać w bardziej zwięzłej formie za pomocą wyrażenia warunkowego:

Wyrażenie warunkowe

```
egzamin = "zaliczony" if liczba_punktow > 60 else "niezaliczony"
```

Zagnieżdżone instrukcje warunkowe *if-else*

- Można tworzyć **warunki złożone** poprzez zagnieżdżenie instrukcji warunkowej **if**



Zagnieżdzona instrukcja warunkowa *if-else*

Przykład użycia instrukcji warunkowej

```
>>> wiek = int(input('Podaj swój wiek: '))
Podaj swój wiek: 20
>>> if not(0 <= wiek <= 150):
...     print('Nie możesz mieć tylu lat')
... else:
...     if wiek < 18:
...         print('Jesteś osobą niepełnoletnią')
...     else:
...         print('Jesteś osobą dorosłą')
...
Jesteś osobą dorosłą
```

- To prowadzi do wielu zagnieżdzonych sekcji

Instrukcja warunkowa *if-elif-else*

- Aby uniknąć wielu poziomów zagnieżdżeń można posłużyć się słowem kluczowym **elif**

Przykład użycia instrukcji warunkowej

```
>>> wiek = int(input('Podaj swój wiek: '))
Podaj swój wiek: 20
>>> if not(0 <= wiek <= 150):
...     print('Nie możesz mieć tylu lat')
... elif wiek < 18:
...     print('Jesteś osobą niepełnoletnią')
... else:
...     print('Jesteś osobą dorosłą')
...
Jesteś osobą dorosłą
```

- Sekcji **elif** może być **dowolnie dużo**

Instrukcja dopasowania

- Składnia instrukcji dopasowania:

Instrukcja dopasowania

```
match subject:  
  case <pattern_1>:  
    <action_1>  
  case <pattern_2>:  
    <action_2>  
  case <pattern_3>:  
    <action_3>  
  case _:  
    <action_wildcard>
```

- Instrukcja jest dostępna od wersji 3.10

Instrukcja dopasowania

Przykład

```
match swiatlo_dla_pieszego:  
    case 'zielone':  
        idz()  
    case 'żółte':  
        opusc_przejscie()  
    case 'czerwone':  
        czekaj()
```

- Wartości wzorców można grupować za pomocą znaku |

Ćwiczenia/przykłady

- Ćwiczenie/przykład 2.5:
BMI – Body Mass Index



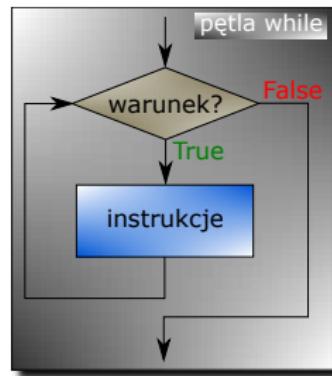
ĆWICZENIA

Pętle

- **Pętla (loop)** – struktura iteracyjna, która umożliwia **wielokrotne wykonanie tego samego bloku kodu**
- W Pythonie mamy dwa rodzaje pętli:
 - pętlę **while**
 - pętlę **for**

Pętla `while`

- Pętla `while` powtarza instrukcję lub grupę instrukcji wielokrotnie, tak długo, gdy podany warunek jest prawdziwy
- Warunek jest sprawdzany **przed** każdym wykonaniem ciała pętli
- Pętla kończy się, gdy warunek stanie się fałszywy



Składnia pętli `while`

```
while expression:  
    statement(s)
```

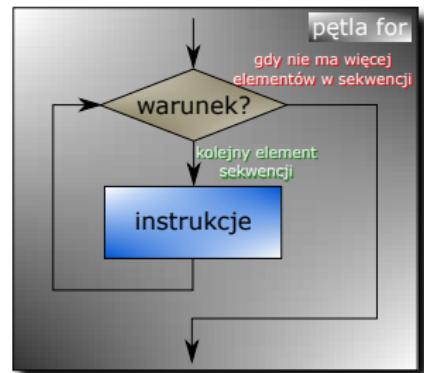
Pętla `while` – przykład

Przykład użycia pętli `while`

```
>>> count = 3      # licznik pętli
>>> while count > 0:
...     print(count)
...     count = count - 1
...
3
2
1
```

Pętla for

- Pętla **for** jest najbliższą instrukcją *for-each* znanej z języka PHP lub Javy
- Umożliwia wykonanie sekwencji operacji na elementach podanego zbioru wartości



Składnia pętli for

```
for item in iterable:  
    statement(s)
```

Pętla *for* – przykład

Przykład użycia pętli *for*

```
>>> for count in [3, 2, 1]:  
...     print(count)  
...  
3  
2  
1
```

Sekcja `else` pętli

- Pętle (`while` oraz `for`) można uzupełnić o opcjonalną sekcję `else`
- Jest ona wykonywana tylko wtedy, gdy pętla zakończy się w sposób “naturalny” – na skutek:
 - wyczerpania się iterowanych elementów
 - osiągnięcia przez testowany warunek fałszywej wartości

Sekcja `else` pętli

Wykorzystanie sekcji `else` pętli

```
>>> count = 3      # licznik pętli
>>> while count > 0:
...     print(count)
...     count -= 1
... else:
...     print('Start')
...
3
2
1
Start
```

Przerywanie pętli

- Wykonywanie pętli można **przedwcześnie zakończyć** poprzez wywołanie instrukcji **break**
- Instrukcja **continue** umożliwia **przerwanie wykonywania bieżącej iteracji** i rozpoczęcie kolejnej
- W przypadku użycia instrukcji **break** nie wykona się kod sekcji **else** pętli

Instrukcja `pass`

- Instrukcja `pass` – instrukcja pusta
- Wykorzystywana wszędzie tam, gdzie zgodnie ze składnią konieczna jest instrukcja, ale nie chcemy, aby był wykonany jakikolwiek kod
- W przypadku klas, aby zagwarantować niepuste ciało, zamiast użycia instrukcji `pass`, zalecane jest wstawienie *docstring'a*

Ćwiczenia/przykłady

- Ćwiczenie/przykład 2.6:
Wieczny kalendarz



ĆWICZENIA

Plan szkolenia

- 1 WPROWADZENIE DO JĘZYKA PYTHON
- 2 PODSTAWOWE KONCEPCJE
- 3 ZŁOŻONE TYPY DANYCH**
- 4 PROGRAMOWANIE FUNKCYJNE
- 5 KLASY I OBIEKTY
- 6 MODUŁY I PAKIETY
- 7 OPERACJE NA PLIKACH
- 8 WYJĄTKI
- 9 WAŻNE WBUDOWANE MODUŁY I BIBLIOTEKI

3 ZŁOŻONE TYPY DANYCH

- typ tekstowy (łańcuchy znaków)
- formatowanie łańcuchów znaków
- operacje na tekstach
- krotki
- zakresy
- listy
- dostęp do elementów sekwencji
- operacje na sekwencjach
- zbiory
- operacje na zbiorach
- słowniki
- operacje na słownikach



Typ tekstowy

- Dane tekstowe w Pythonie są **niemutowalnymi ciągami znaków Unicode** – tworzą **łańcuchy znakowe (strings)**
- Są reprezentowane przez typ *str*
- W Pythonie nie ma odrębnego typu reprezentującego pojedyncze znaki (brak jest typu znakowego)
- Zmienne typu tekstowego można utworzyć za pomocą literałów lub dokonać konwersji z innego typu za pomocą funkcji *str*

Typ tekstowy

- **Literały tekstowe** mogą być ograniczone:
 - apostrofami
 - cudzysłowami
 - potrójnymi apostrofami lub cudzysłowami
- Literały tekstowe ujęte w potrójne ograniczniki mogą obejmować wiele linii – występujące tam znaki białe stają się częścią literału
- Tego typu literały wykorzystuje się do **dokumentowania kodu**

Typ tekstowy – przykłady

Przykłady - literały tekstowe

```
>>> s = 'Python is cool'  
>>> print(s)  
Python is cool  
>>> s = "Python is cool"  
>>> print(s)  
Python is cool  
>>> s = 'Python \  
... is cool'  
>>> print(s)  
Python is cool  
>>> s = '''Python  
... is cool'''  
>>> print(s)  
Python  
is cool  
  
>>> type(s)  
<class 'str'>
```

Typ tekstowy

- W literale tekstowym nie mogą wystąpić znaki będące ogranicznikami tekstu, chyba, że zostaną zamaskowane (*escaped*) za pomocą odwrotnego ukośnika \

Maskowanie znaków specjalnych

```
>>> s1 = "I'm a Python fanatic"  
>>> print(s1)  
I'm a Python fanatic  
  
>>> s2 = 'I\'m a Python fanatic'  
>>> print(s2)  
I'm a Python fanatic
```

Sekwencje znaków specjalnych

SEKWENCJA	ZNACZENIE	KOD ASCII/ISO
\<nowa-linia>	pomiń koniec linii	
\\"	odwrotny ukośnik (<i>backslash</i>)	0x5C
\'	apostrof	0x27
\"	cudzysłów	0x22
\a	<BEL>	0x07
\b	<BS>	0x08
\f	<FF>	0x0C
\n	<LF>	0x0A
\r	<CR>	0x0D
\t	<TAB>	0x09
\v	<VT>	0x0B

Sekwencje znaków specjalnych – pytanie kontrolne

- Co będzie wynikiem poniższego polecenia?



Znaki specjalne

```
>>> print('abnormal', 'a\bnormal')
```

Znaki Unicode

Konwersja: znak -> kod Unicode znaku (funkcja *ord*)

```
>>> s = 'π'  
>>> unicode = ord(s)  
>>> print(unicode, hex(unicode))  
960 0x3c0
```

Konwersja: kod Unicode znaku -> znak (funkcja *chr*)

```
>>> s1 = chr(960)  
>>> print(s1)  
π  
  
>>> s2 = '\u03c0'  
>>> print(s2)  
π
```

Długie literały tekstowe

- Do złączenia literałów w kilku wierszach w jeden długi tekst można użyć znaków kontynuacji oraz operatora konkatenacji:

Tworzenie długiego literała tekstu

```
>>> s = 'This is the way to join '\
...     + 'two long strings together'
>>> print(s)
This is the way to join two long strings together
```

Długie literały tekstowe

- ... lub nawiasów okrągłych do utworzenia pojedynczego wyrażenia:

Tworzenie długiego literała tekstowego

```
>>> s = ('This is the way to join '
...           'two long strings together')
>>> print(s)
This is the way to join two long strings together
```

- Jest to zalecana praktyka

Formatowanie tekstu

- Do sformatowania tekstu można użyć metody *format*:

Funkcja *format*

```
>>> s = 'Good morning, {}. How are you today?'.format('John')

>>> print(s)
Good morning, John. How are you today?
```

Formatowanie tekstu

- Literałami tekstowymi poprzedzonymi przedrostkiem `f` mogą odwołać się do zmiennych znajdujących się w ich zasięgu (od Pythona 3.6)

Formatowanie tekstu

```
>>> name = 'John'  
>>> s = f'Good morning, {name}. How are you today?'  
  
>>> print(s)  
Good morning, John. How are you today?
```

Wybrane metody klasy str

METODA	KATEGORIA
<code>s.capitalize()</code> <code>s.lower()</code> <code>s.upper()</code> <code>s.swapcase()</code> <code>s.title()</code>	zarządzanie wielkością liter
<code>s.ljust(width[, fillchar])</code> <code>s.center(width[, fillchar])</code> <code>s.rjust(width[, fillchar])</code> <code>s.zfill(width)</code>	pozycjonowanie tekstu
<code>s.lstrip([chars])</code> <code>s.strip([chars])</code> <code>s.rstrip([chars])</code>	usuwanie znaków z początku/końca
<code>s.find(sub[, start[, end]])</code> <code>s.rfind(sub[, start[, end]])</code> <code>s.index(sub[, start[, end]])</code> <code>s.rindex(sub[, start[, end]])</code>	wyszukiwanie tekstu
<code>s.split(sep=None, maxsplit=-1)</code> <code>s.rsplit(sep=None, maxsplit=-1)</code> <code>s.splitlines([keepends])</code> <code>s.partition(sep)</code> <code>s.rpartition(sep)</code>	podział tekstu

Wybrane metody klasy str

METODA	KATEGORIA
s.replace(old, new[, count]) s.translate(table)	zamiana znaków
s.join(iterable)	łączenie tekstów
s.isalnum() s.isalpha() s.isdecimal() s.isdigit() s.isnumeric() s.isidentifier() s.islower() s.isupper() s.isprintable() s.isspace() s.istitle()	metody informacyjne
s.startswith(prefix[, start[, end]]) s.endswith(suffix[, start[, end]])	

Ćwiczenia/przykłady



- Ćwiczenie/przykład 3.1:
Skrót

ĆWICZENIA

Krotki

- **Krotka** (*tuple*) – uporządkowana **sekwencja** elementów
- Krotki są **niemutowalne**, więc nie można zmieniać, ani usuwać ich elementów
- Można tworzyć krotki obiektów mutowalnych, ale w ogólności nie jest to dobrą praktyką
- Krotki reprezentują typ *tuple*

Krotki

- Aby utworzyć krotkę, należy jej elementy odseparować **przecinkami**
- Przecinek można umieścić także za ostatnim elementem
- Użycie przecinka jest niezbędne przy tworzeniu krotki jednoelementowej (singletona)

Przykłady krotek

```
>>> t = 1,  
>>> print(t)  
(1,)  
  
>>> t = 'abc', 123  
>>> print(t)  
('abc', 123)
```

Krotki

- Całą zawartość krotki można opcjonalnie ująć **w nawiasy okrągłe**
- Ujęcie pojedynczych elementów w nawiasy nie ma żadnego efektu
- Są one obowiązkowe tylko wtedy, gdy trzeba uniknąć składniowej niejednoznaczności
- Nawiasy są także niezbędne przy utworzeniu krotki pustej
- Krotki dwuelementowe są nazywane **parami** (*pairs*)

Krotki – przykłady

Przykłady krotek

```
>>> t = ()  
>>> print(t)  
()  
  
>>> t = ('abc', 123)  
>>> print(t)  
('abc', 123)  
  
>>> t = (('abc'), (123))  
>>> print(t)  
('abc', 123)  
  
>>> t = (('abc',), (123))  
>>> print(t)  
(('abc',), 123)
```

Ćwiczenia/przykłady

- Ćwiczenie/przykład 3.2:
Wieczny kalendarz



ĆWICZENIA

Zakresy

- **Zakres (range) – niemutowalna sekwencja liczb**
- Do utworzenia zakresu wykorzystuje się funkcję *range*

Funkcja *range*

```
range(stop)
range(start, stop[, step])      # start - inclusive
                                # stop  - exclusive
```

- Argumentami funkcji powinny być liczby typu *int* lub obiekty implementujące specjalną metodę __index__
- Wartości domyślne:

dla parametru <i>start</i>	0
dla parametru <i>step</i>	1

Zakresy

- Zaletą korzystania z zakresów są niewielkie wymagania pamięciowe – zapamiętywana jest wartość początkowa, końcowa i krok
- Zajętość pamięci jest niewielka i niezależna od rozmiaru zakresu
- W miarę potrzeby kolejne wartości zakresu są wyliczane

Użycie zakresów

```
>>> for x in range(3):
...     print(x)
...
0
1
2
>>>
```

Ćwiczenia/przykłady

- Ćwiczenie/przykład 3.3:
Liczba “pechowych” dni
- Ćwiczenie/przykład 3.4:
Losowanie lotto



ĆWICZENIA

Typ listy

- **Lista (*list*)** – uporządkowana **sekwencja** elementów, najczęściej jednorodnych pod względem typu
- Listy są **mutowalne** (inaczej niż teksty i krotki)
- Są reprezentowane przez typ *list*
- Do ich utworzenia można stosować literały – elementy separuje się **przecinkami**, a całość ujmuje się **w nawiasy kwadratowe**
- Za ostatnim elementem listy może opcjonalnie znaleźć się przecinek

Listy – przykład

Przykłady list

```
>>> mylist = []
>>> print(mylist)
[]

>>> mylist = [123]
>>> print(mylist)
[123]

>>> mylist = ['a', 'b', 'c', ]
>>> print(mylist)
['a', 'b', 'c']

>>> print(type(mylist))
<class 'list'>
```

Operacje na listach

METODA	DZIAŁANIE
<code>L.append(x)</code>	dodaje element x do listy L
<code>L.count(x)</code>	zlicza, ile razy element x występuje w liście L
<code>L.extend(m)</code>	dodaje na końcu listy wszystkie elementy iterowalnego obiektu m ten sam efekt można uzyskać za pomocą wyrażenia $L += m$
<code>L.index(x, start, end)</code>	zwraca indeks pierwszej pozycji elementu x w liście L lub jej wycinku w przeciwnym razie zgłasza wyjątek <i>ValueError</i>
<code>L.insert(i, x)</code>	wstawia element x do listy L na pozycji i tekstu
<code>L.pop()</code>	zwraca i usuwa z listy L ostatni element
<code>L.pop(i)</code>	zwraca i usuwa z listy L element na pozycji i
<code>L.remove(x)</code>	usuwa pierwsze wystąpienie elementu x z listy L zgłasza wyjątek <i>ValueError</i> , jeśli element x nie zostanie znaleziony
<code>L.reverse()</code>	odwraca porządek elementów listy L (w miejscu)
<code>L.sort(...)</code>	sortuje listę L (w miejscu) argumenty takie, jak w funkcji <i>sorted</i>

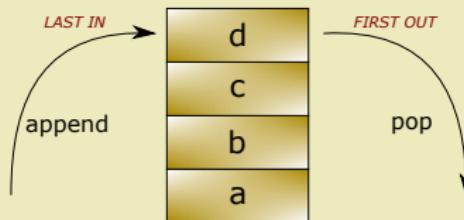
☞ więcej na temat funkcji *sorted*...

Użycie listy jako stosu

- W oparciu o listy można zrealizować strukturę **stosu** (*stack*), tzn. kolejkę LIFO

Realizacja stosu

```
>>> stack = ['a', 'b', 'c']
>>> print(stack)
['a', 'b', 'c']
>>> stack.append('d')
>>> print(stack)
['a', 'b', 'c', 'd']
>>> stack.pop()
'd'
>>> print(stack)
['a', 'b', 'c']
```



- Operacje dodawania i usuwania elementów z końca listy są efektywne

Użycie listy jako kolejki

- Tworzenie zwykłych **kolejek** (*queues*), tzn. kolejek FIFO w oparciu o listy jest również możliwe, ale operacje dodawania i usuwania elementów z początku listy są mało wydajne
- W takim przypadku lepiej rozważyć użycie klasy *deque* z modułu *collections*

Sekwencje

- **Sekwencje (sequences)** – reprezentują skończone, **uporządkowane kontenery** dla obiektów (dokładniej: dla referencji do obiektów)
- Podstawowymi typami sekwencji są:
 - krotki
 - listy
 - zakresy
 - łańcuchy znakowe

Obiekty iterowalne

- Wszystkie sekwencje są **iterowalne**
- **Obiekty iterowalne** (*iterables*) posiadają metodę *__iter__*, która zwraca obiekt iteratora (można też użyć funkcję *iter*)
- **Iterator** umożliwia sekwencyjny dostęp do elementów obiektu iterowalnego – za pomocą metody *__next__* (można też użyć funkcję *next*)

Obiekty iterowalne

- Do elementów obiektów iterowalnych można dotrzeć także za pomocą pętli **for**:

Iteracja za pomocą pętli

```
>>> for znak in 'abc':  
...     print(znak)  
...  
a  
b  
c
```

Obiekty iterowalne

- W oparciu o obiekty iterowalne można tworzyć m.in. krotki (funkcja *tuple*) i listy (funkcja *list*):

Funkcje *tuple* i *list*

```
>>> krotka = tuple('abc')
>>> print(krotka)
('a', 'b', 'c')
```

```
>>> lista = list('abc')
>>> print(lista)
['a', 'b', 'c']
```

Dostęp do elementów sekwencji

- Do kolejnych elementów sekwencji można się dostać podając właściwy **indeks** ($seq[i]$) to i-ty element sekwencji seq)
- Indeksy są numerowane **od zera** i nie mogą być większe niż $N - 1$ (N – długość sekwencji)
- Ujemny indeks i wskazuje element dostępny pod indeksem $i + N$
- Indeksy nie mogą być mniejsze niż $-N$

czyli: $-N \leq \text{indeks} \leq N - 1$

- Jeżeli sekwencja seq składa się z N elementów, to:

$seq[0]$ oraz $seq[-N]$ jest **pierwszym** elementem sekwencji

$seq[N-1]$ oraz $seq[-1]$ jest **ostatnim** elementem sekwencji

Dostęp do elementów sekwencji – przykład

Łańcuch znakowy

```
s = 'Hello world'
```



Wycinki sekwencji (*slicing*)

- Z sekwencji można “wyciąć” (*slicing*) podsekwencję – sekwencję elementów spod wskazanego zakresu indeksów (włącznie z pierwszym, ale bez ostatniego)

seq[start_incl:stop_excl]

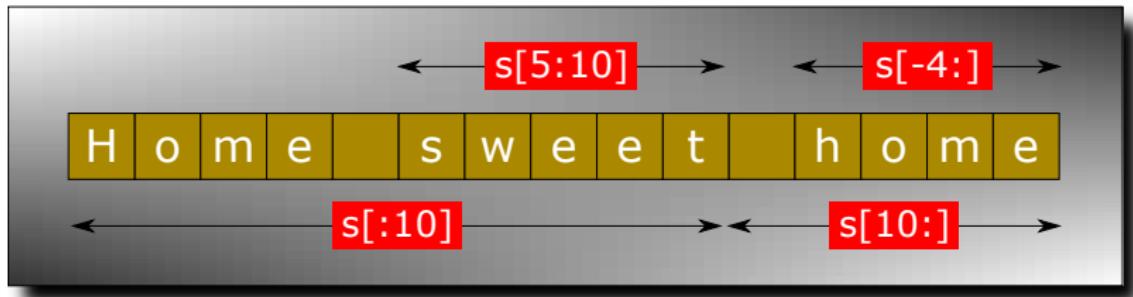
- Pominiecie pierwszego indeksu lub podanie wartości mniejszej niż $-N$ (gdzie N – rozmiar sekwencji) jest równoważne z podaniem wartości 0
- Pominiecie drugiego indeksu lub podanie wartości większej niż N jest równoważne z podaniem wartości N

Wycinki sekwencji (*slicing*) – przykład

- Przykładowo dla sekwencji: $x = ['a', 'b', 'c', 'd', 'e']$

NOTACJA	ZNACZENIE	WARTOŚĆ
$x[1:4]$	element drugi, trzeci i czwarty	$['b', 'c', 'd']$
$x[1:]$	element drugi i kolejne do ostatniego	$['b', 'c', 'd', 'e']$
$x[:-1]$	element pierwszy i kolejne do przedostatniego	$['a', 'b', 'c', 'd']$
$x[:]$	wszystkie elementy (kopia sekwencji)	$['a', 'b', 'c', 'd', 'e']$

Wycinki sekwencji (*slicing*) – przykład



Wycinki sekwencji (*striding*)

- Składnię wycinka sekwencji można uzupełnić o trzeci parametr określający krok – zwracane nie są kolejne elementy sekwencji, tylko co n-ty (*striding*)

seq[start_incl:stop_excl:step]

- Domyślną wartością kroku jest 1
- Wartość zerowa kroku jest niedozwolona

Wycinki sekwencji (*striding*)

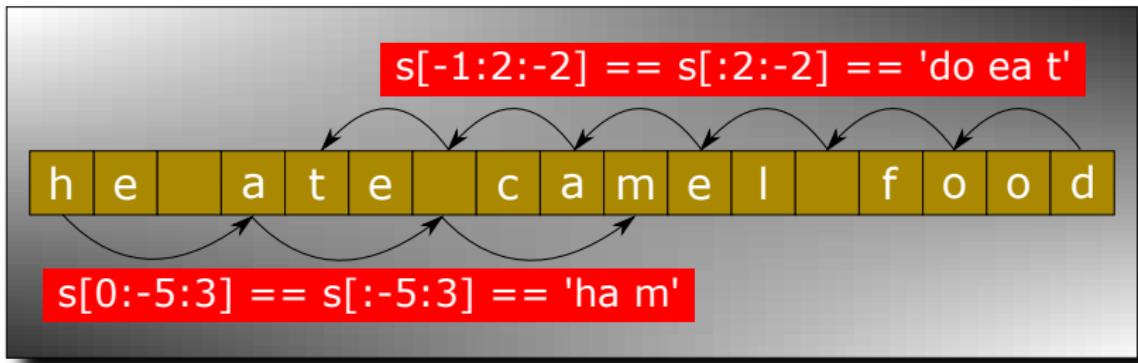
- Jeśli indeks startowy zostanie pominięty, to jego wartością domyślną będzie:

jeśli krok będzie dodatni	0
jeśli krok będzie ujemny	-1

- Jeśli indeks końcowy zostanie pominięty, to jego wartością domyślną będzie:

jeśli krok będzie dodatni	długość sekwencji
jeśli krok będzie ujemny	wartość wskazująca położenie przed pierwszym elementem sekwencji

Wycinki sekwencji (*striding*) – przykład



Wypakowywanie sekwencji

- Zawartość sekwencji można **wypakować** (*unpacking assignment*) przypisując jej zawartość do zestawu zmiennych:

Wypakowanie zawartości krotki

```
>>> m = 'maj', 31
>>> print(m)
('maj', 31)

>>> miesiac, dni = m
>>> print(miesiac)
maj
>>> print(dni)
31
```

- Zmienne *miesiac* i *dni* są **celami** (*targets*) operacji wypakowania

Wypakowywanie sekwencji

- Aby w trakcie wypakowania pominąć wybrane składowe, można jako identyfikatora użyć znaku podkreślenia _

Ignorowanie wybranych składowych

```
>>> x, _, _, y = (1, 2, 3, 4)
>>> print(x)
1
>>> print(y)
4
```

Wypakowywanie sekwencji

- W Pythonie 3 jeden z celów wypakowania można poprzedzić `*`, czyli **operatorem wypakowania sekwencji** (*sequence unpacking operator*)
- Do wyrażenia postaci `*target` (*starred expression*) zostanie przypisana lista elementów sekwencji, które **nie stały się celami innych zmiennych**

Wypakowywanie sekwencji – przykład

Wypakowywanie elementów sekwencji

```
>>> lista = [1, 2, 3, 4, 5]
>>> pierwszy, *srodkowe, ostatni = lista
>>> print(pierwszy)
1
>>> print(srodkowe)
[2, 3, 4]
>>> print(ostatni)
5
```

- Jest to równoważne wypakowaniu:

Wypakowywanie elementów sekwencji

```
pierwszy, srodkowe, ostatni = lista[0], lista[1:-1], lista[-1]
```

Ćwiczenia/przykłady



- Ćwiczenie/przykład 3.5:
Robot

ĆWICZENIA

Funkcja *enumerate*

- Kolejne, zwracane przez funkcję *enumerate* elementy są krotkami zawierającymi:
 - bieżący numer (liczony od wartości *start* – domyślnie od 0)
 - kolejny zwracany przez iterator element

Przykład użycia enumeracji

```
>>> pory_roku = ['wiosna', 'lato', 'jesień', 'zima']

>>> print(list(enumerate(pory_roku)))
[(0, 'wiosna'), (1, 'lato'), (2, 'jesień'), (3, 'zima')]

>>> print(list(enumerate(pory_roku, start = 1)))
[(1, 'wiosna'), (2, 'lato'), (3, 'jesień'), (4, 'zima')]
```

Funkcja `enumerate` – przykład

Przykład użycia enumeracji

```
>>> pory_roku = ['wiosna', 'lato', 'jesień', 'zima']
>>> for nr, pora in enumerate(pory_roku):
...     print(nr, pora)
...
0 wiosna
1 lato
2 jesień
3 zima
```

Typowe operacje na sekwencjach

OPERATOR	SPOSÓB UŻYCIA	DZIAŁANIE
in	x in seq	test, czy element x znajduje się w sekwencji seq
not in	x not in seq	test, czy element x nie występuje w sekwencji seq
+	seq1 + seq2	złączenie (<i>concatenation</i>) sekwencji
*	seq * n, n * seq	n-krotne powielenie sekwencji
[]	seq[i] seq[i:j] seq[i:j:k]	element o indeksie <i>i</i> wycinek sekwencji pomiędzy indeksami <i>i</i> oraz <i>j</i> (<i>slicing</i>) wycinek sekwencji pomiędzy indeksami <i>i</i> oraz <i>j</i> z krokiem co <i>k</i> (<i>striding</i>)

FUNKCJA/METODA	DZIAŁANIE
len(seq)	długość sekwencji seq
min(seq)	najmniejszy element sekwencji seq
max(seq)	największy element sekwencji seq
seq.index(x[, i[, j]])	indeks pierwszego wystąpienia elementu x w sekwencji seq (począwszy od indeksu <i>i</i> i przed indeksem <i>j</i>)
seq.count(x)	całkowita liczba wystąpień elementu x w sekwencji seq

Operacje na sekwencjach – przykłady

Przykłady operacji na sekwencjach

```
>>> print(3 * 'pra' + 'dziadek')  
praprapradziadek
```

```
>>> print('ka' * 2 + 'o')  
kakao
```

```
>>> print('tykwa' in 'antykwariat')  
True
```

Wyrażenia listowe

- **Wyrażenia listowe** (*list comprehensions*) pozwalają w zwięzły sposób odwzorować pewną listę na inną, wykonując na każdym jej elemencie pewną funkcję
- W wyrażeniach listowych wykorzystuje się pętlę **for** do generowania elementów nowej listy
- W najprostszej postaci wyrażenie listowe ma następującą składnię:

Uproszczona składnia wyrażenia listowego

```
[expression for item in iterable]
```

Wyrażenia listowe – przykład

Przykład wyrażenia listowego

```
>>> list1 = [2 * x for x in range(10)]  
  
>>> print(list1)  
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Wyrażenia listowe – przykład

- W wyrażeniu listowym można opcjonalnie użyć wyrażenia logicznego:

Składnia wyrażenia listowego

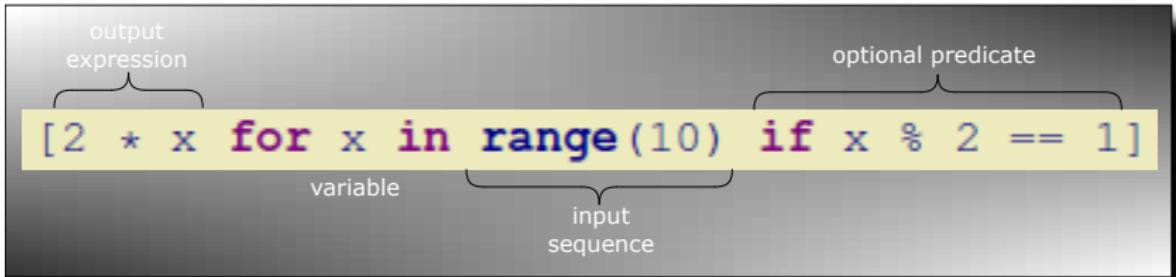
```
[expression for item in iterable if condition]
```

- Składnia jest równoważna konstrukcji:

Odpowiednik wyrażenia listowego

```
>>> temp = []
>>> for item in iterable:
...     if condition:
...         temp.append(expression)
```

Wyrażenia listowe – przykład



Przykład wyrażenia listowego

```
>>> list2 = [2 * x for x in range(10) if x % 2 == 1]  
>>> print(list2)  
[2, 6, 10, 14, 18]
```

Ćwiczenia/przykłady

- Ćwiczenie/przykład 3.6:
Skrót
- Ćwiczenie/przykład 3.7:
Palindrom
- Ćwiczenie/przykład 3.8:
Początki kwartałów



Macierze

- Za pomocą zagnieżdżonych list można w Pythonie zaimplementować **macierze**
- Lista, której elementami są listy tej samej długości tworzy dwuwymiarową macierz (niekoniecznie numeryczną), np.:

$$\begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{bmatrix}$$

Macierz 3x3

```
>>> matrix = [[11, 12, 13], [21, 22, 23], [31, 32, 33]]  
>>> print(matrix)  
[[11, 12, 13], [21, 22, 23], [31, 32, 33]]
```

Spłaszczenie macierzy – zagnieżdżone pętle

- Do “spłaszczenia” macierzy można wykorzystać **zagnieżdżone pętle**:

Spłaszczenie macierzy 3x3

```
>>> f1 = []
>>> for row in matrix:
...     for element in row:
...         f1.append(element)
...
>>> print(f1)
[11, 12, 13, 21, 22, 23, 31, 32, 33]
```

Spłaszczenie macierzy – zagnieżdżone wyrażenia listowe

- Ten sam efekt “spłaszczenia” macierzy można uzyskać wykorzystując **zagnieżdżone wyrażenia listowe**
- Konstrukcja jest bardziej zwarta:

Spłaszczenie macierzy 3x3

```
>>> f2 = [element for row in matrix for element in row]  
>>> print(f2)  
[11, 12, 13, 21, 22, 23, 31, 32, 33]
```

Ćwiczenia/przykłady

- Ćwiczenie/przykład 3.9:
Kwartaly



ĆWICZENIA

- **Zbiory** (*sets*) – iterowalne kolekcje **unikalnych** obiektów
- Elementy zbiorów są **nieuporządkowane** – nie zachowują kolejności elementów podczas iteracji
- W stosunku do zbiorów można stosować:
 - operator `in`
 - operatory porównania
 - operatory bitowe

- Elementy zbiorów mogą być różnych typów, ale muszą być **haszowalne** (*hashable*), tzn. definiować **metodę mieszającą** `_hash_` (inaczej: funkcję skrótu)
- Jej wartość **nie może ulec zmianie w cyklu życia obiektu**, więc obiekty muszą być niemutowalne
- Do sprawdzenia równości obiektów wykorzystuje się specjalną metodę `_eq_`
- Jeśli dwa obiekty są równe, to ich funkcje skrótu również muszą zwrócić identyczne wartości

Typy zbiorów

- Python definiuje dwa typy zbiorów:

<i>set</i>	zbiory mutowalne
<i>frozenset</i>	zbiory niemutowalne

- Wszystkie niemutowalne, wbudowane typy danych (np. *int*, *float*, *tuple*, *str*) **są haszowalne** i mogą być elementami innych zbiorów
- Wbudowane typy mutowalne (np. *list*) **nie są haszowalne**, gdyż wartość funkcji skrótu ulega zmianie w zależności od elementów, które zawierają

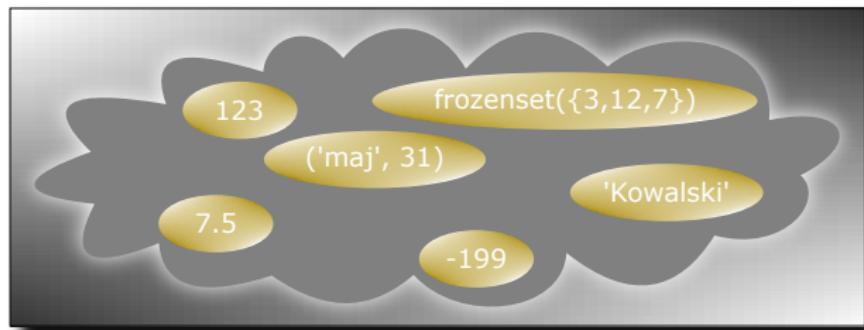
Zbiory typu *set*

- Zbiór typu *set* – nieuporządkowana kolekcja dowolnej liczby (także zerowej) referencji odnoszących się do haszowalnych obiektów
- Tego typu zbiory **są mutowalne**, a więc można do nich dodawać i usuwać elementy
- Zbiory **są nieuporządkowane**, a więc nie są indeksowane i nie można tworzyć ich wycinków
- Niepuste zbiory typu *set* można utworzyć za pomocą literałów – elementy zbioru trzeba odseparować **przecinkami** i całość ująć **w nawiasy klamrowe {}**

Zbiory typu set

Zbiór typu *set*

```
>>> zbior = {-199, 7.5, 'Kowalski', 123, ('maj', 31),  
...           frozenset({3, 12, 7})}  
>>> print(zbior)  
{('maj', 31), 7.5, frozenset({3, 12, 7}), 'Kowalski', -199, 123}
```



Zbiory typu set

- Do utworzenia zbioru **set** (oprócz literałów) można użyć funkcji:

Funkcja **set**

```
set ([iterable])
```

- Jest to jedyny sposób na utworzenie **pustego zbioru** (nie da się tego zrobić za pomocą literałów)
- Podanie jako argumentu innego zbioru, utworzy jego płytka kopię
- W przypadku podania obiektu innego typu nastąpi próba jego konwersji na zbiór

Zbiory – przykłady

Przykłady zbiorów typu *set*

```
>>> set1 = {1, 2, 3}
>>> print(set1)
{1, 2, 3}

>>> print(type(set1))
<class 'set'>

>>> set2 = set()
>>> print(set2)
set()

>>> set3 = set('cool')
>>> print(set3)
{'c', 'l', 'o'}
```

Unikalność elementów zbioru

- Zbiory przechowują wartości unikalne
- Dodanie elementu-duplikatu nie zmienia zawartości zbioru
- Poniższe zbiory są identyczne:

Unikalność elementów zbioru

```
>>> s1 = set('tests')
>>> s2 = set('set')
>>> s3 = {'e', 's', 't'}

>>> print(s1 == s2 == s3)
True
```

Eliminacja duplikatów

- Zbiory można wykorzystać do pozbycia się elementów-duplikatów:

Eliminacja duplikatów

```
>>> list1 = list('coffee')
>>> print(list1)
['c', 'o', 'f', 'f', 'e', 'e']

>>> list2 = list(set(list1))
>>> print(list2)
['e', 'c', 'f', 'o']
```

Operacje na zbiorach

- **Suma zbiorów (union):**

Suma zbiorów

```
>>> print(set('ocean') | set('sea'))  
{'n', 'a', 'c', 's', 'e', 'o'}
```



Operacje na zbiorach

- Część wspólna zbiorów (*intersection*):

Część wspólna zbiorów

```
>>> print(set('ocean') & set('sea'))  
{'a', 'e'}
```



Operacje na zbiorach

- **Różnica** zbiorów (*difference*):

Różnica zbiorów

```
>>> print(set('ocean') - set('sea'))  
{'n', 'c', 'o'}
```



Operacje na zbiorach

- Różnica symetryczna zbiorów (*symmetric difference*):

Różnica symetryczna zbiorów

```
>>> print(set('ocean') ^ set('sea'))  
{'n', 's', 'c', 'o'}
```



Operacje na zbiorach

METODA	DZIAŁANIE
<code>s.add(x)</code>	dodaje element x do zbioru s , jeśli go nie zawiera
<code>s.clear()</code>	usuwa wszystkie elementy ze zbioru s
<code>s.copy()</code>	zwraca płytka kopię zbioru s
<code>s.difference(t)</code>	zwraca zbiór będący różnicą zbiorów s i t odpowiada wyrażeniu $s - t$
<code>s.difference_update(t)</code>	usuwa ze zbioru s wszystkie elementy występujące w zbiorze t odpowiada operacji $s -= t$
<code>s.discard(x)</code>	usuwa element x ze zbioru s , jeśli znajduje się w zbiorze s
<code>s.intersection(t)</code>	zwraca część wspólną zbiorów s i t odpowiada wyrażeniu $s & t$
<code>s.intersection_update(t)</code>	pozostawia w zbiorze s tylko elementy, które występują jednocześnie w zbiorach s i t odpowiada operacji $s &= t$
<code>s.isdisjoint(t)</code>	testuje, czy zbiory s i t są rozłączne
<code>s.issubset(t)</code>	testuje, czy zbiór s jest równy zbiorowi t lub jest jego podzbiorem odpowiada warunkowi $s <= t$
<code>s.issuperset(t)</code>	testuje, czy zbiór s jest równy zbiorowi t lub jest jego nadzbiorem odpowiada warunkowi $s >= t$

Operacje na zbiorach

METODA	DZIAŁANIE
<code>s.pop()</code>	zwraca i usuwa losowy element ze zbioru s lub zgłasza wyjątek <code>KeyError</code> , jeżeli zbiór jest pusty
<code>s.remove(x)</code>	usuwa element x ze zbioru s lub zgłasza wyjątek <code>KeyError</code> , jeśli tego elementu nie ma w zbiorze
<code>s.symmetric_difference(t)</code>	zwraca różnicę symetryczną zbiorów s i t (zbior elementów występujących tylko w zbiorze s lub tylko w zbiorze t , ale nie w obu jednocześnie) odpowiada wyrażeniu $s \Delta t$
<code>s.symmetric_difference_update(t)</code>	aktualizuje zbiór s , by zawierał różnicę symetryczną zbiorów s i t odpowiada operacji $s \Delta= t$
<code>s.union(t)</code>	zwraca sumę zbiorów s i t odpowiada wyrażeniu $s t$
<code>s.update(t)</code>	uzupełnia zbiór s o elementy zbioru t , które nie występują w s odpowiada operacji $s = t$

- W odróżnieniu od operatorów, metody aktualizujące (`s.update`, `s.intersection_update`, itp.) jako argument przyjmują dowolne obiekty iterowalne (operatorы – tylko zbiory)

Tworzenie zbiorów za pomocą pętli

- Zbiory można zapełnić wartościami dołączając kolejne elementy w pętli:

Tworzenie zbioru

```
>>> sentence = 'to be or not to be'  
>>> words = sentence.split()  
  
>>> initials = set()  
>>> for word in words:  
...     initials.add(word[0])  
...  
>>> print(initials)  
{'t', 'b', 'n', 'o'}
```

Tworzenie zbiorów za pomocą wyrażeń

- Do utworzenia zbioru można użyć **konstrukcji podobnej do wyrażeń listowych** (*set comprehension*):

Tworzenie zbioru

```
>>> sentence = 'to be or not to be'  
>>> words = sentence.split()  
  
>>> initials = {word[0] for word in words}  
>>> print(initials)  
{'t', 'b', 'n', 'o'}
```

Zbiory typu *frozense*t

- Zbiór typu *frozense*t – zbiór, który po utworzeniu **nie może podlegać modyfikacjom** (jest niemutowalny)
- Zbiory tego typu tworzy się za pomocą funkcji:

Funkcja *frozense*t

```
frozense([iterable])
```

- Pominiecie argumentu konstruktora umożliwi utworzenie pustego zbioru
- Podanie jako argumentu zbioru typu *frozense*t, utworzy jego płytka kopię
- W przypadku podania obiektu innego typu nastąpi próba jego konwersji na zbiór *frozense*t

Zbiory typu *frozense*

- Zbiory *frozense* są niemutowalne, a więc wspierają wszystkie te operacje na zbiorach, które ich nie zmieniają
- Tymi metodami są:
 - *s.copy()*
 - *s.difference(t)*
 - *s.intersection(t)*
 - *s.isdisjoint(t)*
 - *s.issubset(t)*
 - *s.issuperset(t)*
 - *s.symmetric_difference(t)*
 - *s.union(t)*

Słowniki

- **Słowniki** (*dictionaries, mappings*) – kolekcje dowolnych obiektów, które są indeksowane za pomocą haszowalnych obiektów zw. **kluczami**
- Słowniki można traktować jak **tablice asocjacyjne** – kolekcje, których elementy mają postać dwuelementowych krotek (par: *unikalny_klucz → wartość*)
- Słowniki są **mutowalne**

Implementacje słowników

- Python 3.0 definiuje **2 nieuporządkowane odwzorowania**:
 - wbudowany typ *dict*
 - typ *defaultdict* z modułu *collections* standardowej biblioteki
- Python 3.1 definiuje **odwzorowanie porządkujące elementy według kolejności wstawiania** (*insertion order*)
 - typ *OrderedDict* z modułu *collections* standardowej biblioteki

Porównywanie

- Słowniki można porównywać za pomocą operatorów `==` oraz `!=`
- Porównania są dokonywane na kolejnych elementach słowników (i rekurencyjnie na elementach zagnieżdżonych, np. elementach krotek, czy słowników)
- Pozostałe operatory porównań (`<`, `<=`, `>`, `>=`) nie są wspierane, gdyż ich użycie nie ma sensu w stosunku do typów nieuporządkowanych

Słowniki typu *dict*

- Typ *dict* – reprezentuje **nieuporządkowane kolekcje** zawierające dowolną (w tym zerową) liczbę par *klucz-wartość*
 - **klucze** – obiekty haszowalne
 - **wartości** – obiekty dowolnego typu
- Słowniki są **mutowalne**, więc można do nich dodawać i usuwać z nich elementy
- Tego typu słowniki są nieuporządkowane, więc nie posiadają indeksów i nie można tworzyć ich wycinków

Słowniki

- Do utworzenia słownika można użyć elementów oddzielonych przecinkami
- Każdy element zawiera **klucz odseparowany od wartości za pomocą dwukropka**
- Całość definicji jest ujęta **w nawiasy klamrowe { }**

- Jeżeli w takim mapowaniu wystąpią pary z takimi samymi kluczami, to obowiązuje **ostatnia** podana definicja

Słowniki

Przykład słownika

```
>>> d = {'imie': 'Jan',
...         'nazwisko': 'Kowalski',
...         'wiek': 25,
...         'praca': 'Altkom',
...         'stanowisko': 'programista Pythona',
...         'adres': ('ul. Chłodna 51', '00-867', 'Warszawa') }
```



- Do utworzenia słownika można użyć funkcji *dict*:

Funkcja *dict*

```
dict(**kwarg)
dict(mapping, **kwarg)
dict(iterable, **kwarg)
```

- Zapis ***kwarg* oznacza możliwość podania dowolnej ilości **argumentów nazwanych** (*keyword arguments*) o postaci *name=value* oddzielonych przecinkami
- Podanie jako argumentu mapowania tworzy nowy słownik z takim samym zestawem kluczy i wartości

Słowniki – przykłady

Przykłady słowników

```
>>> kwartal1 = dict(styczeń=31, luty=28, marzec=31)

>>> kwartal2 = dict([('kwiecień', 30), ('maj', 31),
...                      ('czerwiec', 30)])

>>> kwartal3 = {'lipiec': 31, 'sierpień': 31, 'wrzesień': 30}

>>> polrocze2 = dict(kwartal3,
...                      październik=31,
...                      listopad=30,
...                      grudzień=31
... )

>>> print(polrocze2)
{'lipiec': 31, 'sierpień': 31, 'wrzesień': 30, 'październik': 31,
'listopad': 30, 'grudzień': 31}
```

Tworzenie słowników za pomocą wyrażeń słownikowych

- Podobnie jak listy i zbiory, również słowniki można tworzyć za pomocą **wyrażeń słownikowych** (*dictionary comprehensions*):

Słowniki ang-pl i pl-ang

```
>>> en2pl = {'apple': 'jabłko', 'pear': 'gruszka',
...             'plum': 'śliwka'}
>>> print(en2pl)
{'apple': 'jabłko', 'pear': 'gruszka', 'plum': 'śliwka'}

>>> pl2en = {value: key for key, value in en2pl.items()}
>>> print(pl2en)
{'jabłko': 'apple', 'gruszka': 'pear', 'śliwka': 'plum'}
```

Słowniki

- Do utworzenia słownika może być przydatna wbudowana funkcja `zip`
- Funkcja zwraca iterowalny zbiór krotek
- W kontekście słowników można użyć tej funkcji podając dwa argumenty:
 - iterowalną sekwencję kluczy
 - iterowalną sekwencję wartości

Przykład słownika

```
>>> kwartal4 = dict(zip(['październik', 'listopad', 'grudzień'],
...                      [31, 30, 31]))  
  
>>> print(kwartal4)  
{'październik': 31, 'listopad': 30, 'grudzień': 31}
```

Modyfikacja zawartości słownika

- Nawiązy kwadratowe umożliwiają dotarcie do wartości związanej z danym kluczem

Dostęp do elementów słownika

```
>>> codes = {'A': 65, 'a': 97, '0': 48}  
  
>>> print(codes['a'])  
97  
  
>>> print(codes['b'])  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
KeyError: 'b'
```

Dodawanie elementów do słownika

- W podobny sposób można dodawać elementy do słownika:

Dodawanie elementów do słownika

```
>>> codes = {'A': 65, 'a': 97, '0': 48}  
>>> codes[' '] = 32  
  
>>> print(codes)  
{'A': 65, 'a': 97, '0': 48, ' ': 32}
```

Operacje na słownikach

METODA	DZIAŁANIE
d. clear()	usuwa wszystkie elementy ze słownika <i>d</i>
d. copy()	zwraca płytka kopię słownika <i>d</i>
dict. fromkeys(s, v)	zwraca słownik, którego kluczami są elementy sekwencji <i>s</i> , a wartościami element <i>v</i> (jeśli <i>v</i> nie podano, to wartościami są wartości <i>None</i>)
d. get(k)	zwraca wartość związaną z kluczem <i>k</i> lub wartość <i>None</i> , jeśli takiego klucza nie ma
d. get(k, v)	zwraca wartość związaną z kluczem <i>k</i> lub podaną wartość <i>v</i> , jeśli takiego klucza nie ma
d. pop(k)	zwraca i usuwa ze słownika <i>d</i> wartość związaną z kluczem <i>k</i> , zgłasza wyjątek <i>KeyError</i> , jeśli klucza nie ma w słowniku
d. pop(k, v)	zwraca i usuwa ze słownika <i>d</i> wartość związaną z kluczem <i>k</i> , jeśli klucza nie ma w słowniku, to zwraca wartość <i>v</i>
d. popitem()	zwraca i usuwa ze słownika <i>d</i> dowolną parę klucz-wartość zgłasza wyjątek <i>KeyError</i> , jeśli słownik jest pusty
d. setdefault(k, v)	działa tak, jak d. get() jeśli w słowniku nie ma klucza <i>k</i> , to do słownika wstawiana jest para <i>(k, v)</i> i zwracana jest wartość <i>v</i> domyślną wartością dla <i>v</i> jest <i>None</i>
d. update(a)	dodaje do słownika <i>d</i> pary (klucz-wartość), które są w <i>a</i> , a nie ma ich w <i>d</i> , w przeciwnym razie – aktualizuje wartości <i>a</i> może być: słownikiem, obiektem iterowalnym par lub nazwanymi argumentami

Operacje na słownikach

METODA	DZIAŁANIE
d.keys()	zwraca widok wszystkich kluczy słownika <i>d</i>
d.values()	zwraca widok wszystkich wartości słownika <i>d</i>
d.items()	zwraca widok wszystkich par (klucz, wartość) słownika <i>d</i>

- **Widok** jest obiektem iterowalnym, **tylko do odczytu**, zawierającym dane słownika (w zależności od użytej metody: elementy, klucze lub wartości)
- Zmiany dokonane w słowniku są widoczne także w widokach

Przykład użycia widoku

```
>>> osoba = {'imie': 'Jan', 'nazwisko': 'Kowalski'}  
>>> widok = osoba.values()  
>>> osoba['imie'] = 'Artur'  
>>> print(widok)  
dict_values(['Artur', 'Kowalski'])
```

Ćwiczenia/przykłady

- Ćwiczenie/przykład 3.10:
Liczby rzymskie
- Ćwiczenie/przykład 3.11:
“Baza” osób



ĆWICZENIA

Plan szkolenia

- 1 WPROWADZENIE DO JĘZYKA PYTHON
- 2 PODSTAWOWE KONCEPCJE
- 3 ZŁOŻONE TYPY DANYCH
- 4 PROGRAMOWANIE FUNKCYJNE**
- 5 KLASY I OBIEKTY
- 6 MODUŁY I PAKIETY
- 7 OPERACJE NA PLIKACH
- 8 WYJĄTKI
- 9 WAŻNE WBUDOWANE MODUŁY I BIBLIOTEKI

Plan modułu

4 PROGRAMOWANIE FUNKCYJNE

- funkcje – wprowadzenie
- definiowanie funkcji
- parametry funkcji
- funkcje ze zmienną liczbą parametrów
- zasięgi zmiennych i reguła LEGB
- funkcje jako argumenty
- funkcje lambda
- typowanie dynamiczne vs. typowanie statyczne
- dokumentowanie kodu funkcji



Funkcje w Pythonie

- **Funkcja** (*function*) – grupa instrukcji, które można wykonać “na życzenie”
- Funkcje służą do definiowania bloków kodu, które mogą być wielokrotnie wykorzystywane
- Umożliwiają zwiększenie modularności aplikacji

Wywoływanie funkcji

- Wykonanie bloku instrukcji funkcji nazywamy **wywołaniem funkcji** (*function call*)
- Podczas wywołania, do funkcji przekazywane są **argumenty pozycyjne**, stanowiące dane na których funkcja wykonuje operacje
- Wywołanie funkcji jest wyrażeniem o postaci:

Składnia wywołania funkcji

`function_name (arguments)`

Definiowanie funkcji

- Własne funkcje definiuje się z użyciem słowa kluczowego **def**

Składnia definicji funkcji

```
def function_name(parameters):  
    statement(s)
```

<i>function_name</i>	identyfikator funkcji jest zmienną, która zostaje powiązana z obiektem funkcji w momencie wykonania deklaracji def
<i>parameters</i>	opcjonalna lista parametrów funkcji

Definiowanie funkcji

- Funkcja może posiadać dowolną (w tym zerową) liczbę **parametrów formalnych**
- Pełnią one rolę identyfikatorów, które podczas wywołania funkcji są inicjowane podanymi wartościami, tzw. **argumentami pozycyjnymi**
- Jeśli funkcja posiada więcej niż jeden parametr, to do ich separacji używa się przecinka

Ciało funkcji

- Niepusty zestaw instrukcji stanowiących treść funkcji nazywany jest **ciałem funkcji** (*function body*)
- Wykonanie deklaracji **def** (utworzenie obiektu funkcji) nie powoduje wykonania ciała funkcji
- Jest ono wykonywane każdorazowo podczas wywołania funkcji

Przykład definicji funkcji

```
>>> def pole_prostokata(a, b):  
...     return a * b
```

Wartość funkcji

- Każda funkcja w Pythonie **zwraca wartość**, choć dopuszczalne jest (i częste) ignorowanie zwracanej wartości
- Wartość można zwrócić za pomocą instrukcji **return**
- Jeśli w funkcji nie ma tej instrukcji lub nie ma za nią zwracanej wartości, to wartością funkcji jest **None**
- Dobrą praktyką programowania jest pomijanie instrukcji **return**, jeśli za nią nie ma zwracanej wartości

Dynamiczne typowanie

- W definicji funkcji nie określa się typów parametrów
- O tym, czy argument “pasuje” do danego parametru nie decyduje jego typ, tylko to, czy może być **użyty w bieżącym kontekście** (jego zachowanie)
- Test odbywa się **dynamicznie**, w momencie próby użycia argumentu
- Taki rodzaj typowania określa się terminem **duck typing**

Duck typing

*"If it walks like a duck
and it quacks like a duck,
then it must be a duck"*



Duck typing – przykład

Duck typing

```
>>> def calculate(a, b, c):
...     return (a + b) * c
...
>>> print(calculate(1, 2, 3))
9

>>> print(calculate([1, 2, 3], [4, 5, 6], 2))
[1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]

>>> print(calculate('black ', 'and white ', 3))
black and white, black and white, black and white,
```

Parametry funkcji

- **Parametry formalne** – służą do nazwania wartości przekazywanych w wywołaniu funkcji
- Nazwy są przypisywane do **lokalnej przestrzeni nazw** – przestrzeń jest każdorazowo tworzona podczas wywołania funkcji i niszczona przy jej opuszczaniu
- **Parametry pozycyjne** (tzn. te, dla których nie określono wartości domyślnych) są **obowiązkowe** – w każdym wywołaniu funkcji trzeba dostarczyć im wartości

Parametry funkcji

- Niektóre funkcje mogą posiadać parametry, dla których można podać **sensowne wartości domyślne**
- Takie parametry nazywane są **parametrami nazwanymi** (*named parameters, keyword parameters*)
- Mają postać: *identifier=expression*

Parametry funkcji

- Parametry nazwane mogą wystąpić dopiero za wszystkimi parametrami pozycyjnymi (obowiązkowymi)

Funkcja z parametrami nazwanymi

```
>>> def f(x, a=1, b=2):  
...     print(x, a, b)  
...
```

- Argumenty nazwane są **opcjonalne** – w razie ich pominięcia w wywołaniu funkcji, parametry zostaną zainicjowane wartościami domyślnymi

Funkcje z parametrami domyślnymi

- Argumenty wywołania nie muszą być podane **w takiej samej kolejności**, co parametry funkcji – wtedy oprócz wartości trzeba podać nazwy parametrów

Wywołanie funkcji z parametrami nazwanymi

```
>>> print(f(8))  
8 1 2
```

```
>>> print(f(8, 9, 10))  
8 9 10
```

```
>>> print(f(8, b=4))  
8 1 4
```

```
>>> print(f(b=4, a=3, x=8))  
8 3 4
```

Parametry domyślne

- Wartości domyślne parametrów funkcji tworzone są **jednorazowo w momencie wykonania instrukcji `def`**, czyli tworzenia funkcji, a nie w momencie każdorazowego jej wywołania
- Takie zachowanie nie stanowi problemu, jeśli parametry są **niemutowalne** – przy każdym wywołaniu funkcji będą one miały tę samą wartość domyślną

Funkcja z parametrem niemutowalnym

```
>>> def f(x, a=1):  
...     print(x, a)  
  
>>> print(f(4, 5))  
4 5  
>>> print(f(4))  
4 1
```

Parametry domyślne – pytanie kontrolne

Funkcja z parametrem mutowalnym

```
>>> def f(element, lista = []):
...     lista.append(element)
...     print(lista)
...
>>> f(1, [])
[1]
>>> f(2, [])
[2]
>>> f(3)
[3]
>>> f(4)
[3, 4]
```

- Jak można poprawić tę funkcję?



Funkcje ze zmienną liczbą parametrów

- Można tworzyć funkcje ze zmienną (nieokreśloną) liczbą parametrów
- Na liście parametrów mogą wystąpić deklaracje:

*args	reprezentuje krotkę zawierającą nienazwane argumenty, które nie zostały związane z zadeklarowanymi jawnie parametrami
**kwds	reprezentuje słownik zawierający nazwane argumenty, które nie zostały związane z zadeklarowanymi jawnie parametrami

Ćwiczenia/przykłady

- Ćwiczenie/przykład 4.1:
Wieczny kalendarz – użycie funkcji



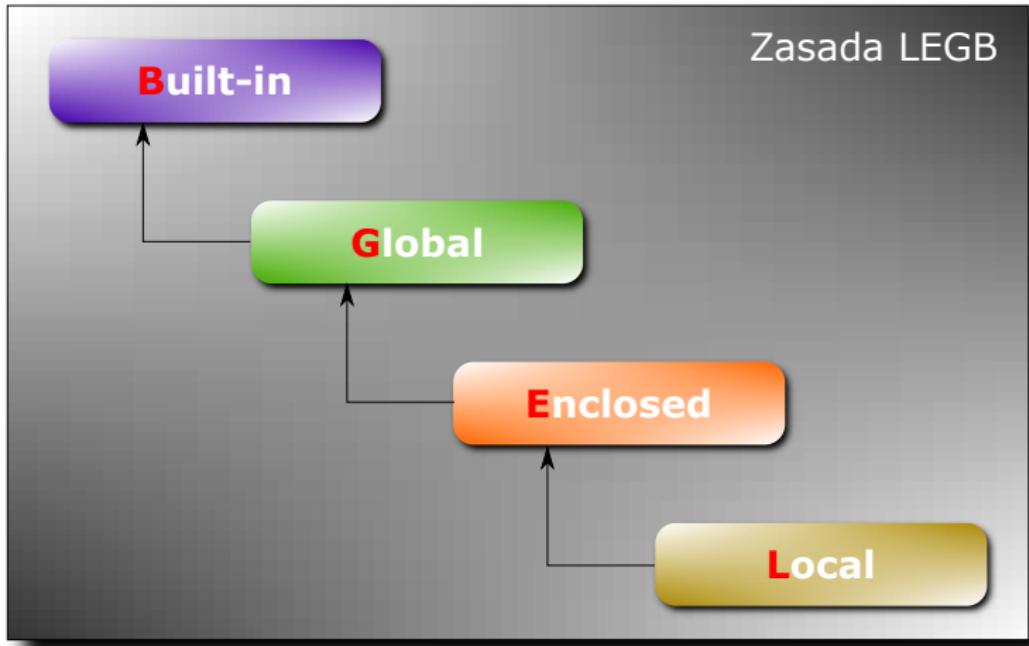
ĆWICZENIA

Przestrzenie nazw i zasięgi

- **Przestrzenie nazw** (*namespaces*) – są kontenerami (o strukturze słowników) przechowującymi odwzorowania *nazwa* → *obiekt*
- Tym samym umożliwiają dostęp do obiektu po nazwie
- W Pythonie może być wiele niezależnych przestrzeni nazw
- Ponieważ w każdej z nich mogą występować klucze o tej samej nazwie, konieczny jest mechanizm jednoznacznego odniesienia do obiektu o danej nazwie
- Reguła wyboru obiektu o danej nazwie spośród obiektów znajdujących się na różnych poziomach hierarchii nosi nazwę **zasady LEGB** (*LEGB rule*)

Zasada LEGB

- Kolejność przeszukiwania zasięgów



Zasięg lokalny

- Parametry funkcji oraz identyfikatory w ciele funkcji, które są związane (poprzez przypisanie lub inne deklaracje, jak np. `def`) tworzą **lokalną przestrzeń nazw funkcji** (*function's local namespace*)
- Jest ona inaczej zwana **lokalnym zasięgiem** (*local scope*)
- Wszystkie zmienne należące do tej przestrzeni są **zmiennymi lokalnymi funkcji** (*local variables*)
- Funkcja `locals` zwraca słownik zawierający nazwy i wartości zmiennych lokalnych

Zasięg globalny

- Zmienne, które nie są lokalne (w przypadku braku funkcji zagnieżdżonych) są **zmiennymi globalnymi**
- Zmienne globalne są atrybutami obiektu modułu
- Funkcja *globals* zwraca słownik zawierający nazwy i wartości zmiennych globalnych

☞ więcej na ten temat...

Zasięg globalny – przykład

- Funkcja może mieć dostęp do zmiennych globalnych

Zmienne globalne

```
>>> a = 111 # zasięg globalny
>>> def f():
...     return a
...
>>> print(f(), '[wartość a wewnętrz f()]')
111 [wartość a wewnętrz f()]
>>>
>>> print(a, '[wartość a na zewnątrz f()]')
111 [wartość a na zewnątrz f()]
```

Zasięg globalny

- W przypadku, gdy zmienna globalna i zmienna lokalna funkcji mają takie same nazwy, to nazwa w ciele funkcji odnosi się do zmiennej lokalnej
- Oznacza to, że zmienna lokalna **przesłania** (*hides*) zmienną globalną → p. zasada LEGB

Przesłanianie zmiennych

```
>>> a = 111      # zasięg globalny
>>> def f():
...     a = 222  # zasięg lokalny
...     return a
...
>>> print(f(), '[wartość a wewnętrz f()]')
222 [wartość a wewnętrz f()]
>>> print(a, '[wartość a na zewnątrz f()]')
111 [wartość a na zewnątrz f()]
```

Zasięg globalny

- Jeśli funkcja chce zmienić przypisanie zmiennej globalnej (a nie jej stan) to musi użyć deklaracji **global**

Zmienna globalna

```
>>> a = 111
>>> def f():
...     global a
...     a += 1
...     return a
...
>>> print(a)
111
>>> print(f())
112
>>> print(a)
112
```

- Taki styl programowania, choć dozwolony, nie jest zalecany

Przekazywanie funkcji

- Funkcje w Pythonie **są obiektami** i tak, jak wszystkie obiekty mogą być:
 - przypisywane do zmiennych
 - argumentami wywołań funkcji
 - elementami kontenerów (elementami list, kluczami i wartościami słowników, itp.)
 - atrybutami obiektów

Przekazywanie funkcji

- Do funkcji jako argument można przekazać:
 - wartość funkcji (wynik wywołania funkcji)
 - obiekt funkcji (kod funkcji)
- W Pythonie jest wiele funkcji, które jako argumentu oczekują podania obiektu innej funkcji
- Można też takie funkcje tworzyć samodzielnie

Wbudowana funkcja *filter*

Funkcja *filter*

```
filter(function, iterable)
```

- Wbudowana funkcja *filter* tworzy iterator z tych elementów *iterable*, dla których funkcja *function* zwraca wartość **True**
- Podanie zamiast funkcji – wartości **None** oznacza, że zostanie zastosowana funkcja tożsamości (*identity function*), w efekcie czego zostaną usunięte te elementy, których kontekstem logicznym będzie **False**

Wbudowana funkcja *filter* – przykład

Przykład użycia funkcji *filter*

```
>>> miesiace = ('styczeń', 'luty', 'marzec', 'kwiecień', 'maj',
...                 'czerwiec', 'lipiec', 'sierpień', 'wrzesień',
...                 'październik', 'listopad', 'grudzień')

>>> def predykat_dlugosci(miesiac):
...     return 5 <= len(miesiac) <= 7

>>> f = filter(predykat_dlugosci, miesiace)

>>> print(*f)
styczeń marzec lipiec
```

Wbudowana funkcja *filter* – pytanie kontrolne

Pytanie kontrolne

```
>>> cyfry = range(10)  
  
>>> print(*cyfry)  
0 1 2 3 4 5 6 7 8 9  
  
>>> f = filter(None, cyfry)  
>>> print(*f)
```

- Jaki będzie wynik wykonania powyższego skryptu?



Wbudowana funkcja *map*

Funkcja *map*

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

- Wbudowana funkcja *map* tworzy iterator do elementów będących wynikami zastosowania funkcji *function* do każdego elementu *iterable*

Wbudowana funkcja *map*

Przykład użycia funkcji *map*

```
>>> zdanie = ('Raz w maju w drugą niedzielę '  
             'Pi liczył cyfry pan Felek')  
>>> slowa = zdanie.split()  
  
>>> def ile_znakow(tekst):  
...     return len(tekst)  
...  
>>> dlugosci = map(ile_znakow, slowa)  
>>> print(*dlugosci)  
3 1 4 1 5 9 2 6 5 3 5
```

Wbudowana funkcja *sorted*

Składnia funkcji *sorted*

```
sorted(iterable, *, key=None, reverse=False)
```

- Wbudowana funkcja *sorted* tworzy nową, posortowaną listę elementów zwracanych przez *iterable*
- Opcjonalne argumenty:

<i>key</i>	jednoargumentowa funkcja służąca do wyodrębnienia z każdego elementu klucza porządkującego domyślna wartość None oznacza porównywanie bezpośrednie
<i>reverse</i>	wartość logiczna umożliwiająca odwrócenie porządku sortowania

Wbudowana funkcja sorted

Sortowanie – klucz prosty

```
>>> metale = [(11, 29, 'Cu'), (1, 3, 'Li'), (11, 79, 'Au'),  
...             (8, 26, 'Fe')]  
... # (grupa, liczba atomowa, symbol)  
  
>>> print(sorted(metale))  
[(1, 3, 'Li'), (8, 26, 'Fe'), (11, 29, 'Cu'), (11, 79, 'Au')]  
  
>>> def wg_symbolu(pierwiastek):  
...     return pierwiastek[2]  
...  
>>> print(sorted(metale, key=wg_symbolu))  
[(11, 79, 'Au'), (11, 29, 'Cu'), (8, 26, 'Fe'), (1, 3, 'Li')]
```

Wbudowana funkcja *sorted*

Sortowanie – klucz złożony

```
>>> def wg_grupy_i_symbolu(pierwiastek):
...     return pierwiastek[0], pierwiastek[2]
...
>>> print(sorted(metale, key=wg_grupy_i_symbolu))
[(1, 3, 'Li'), (8, 26, 'Fe'), (11, 79, 'Au'), (11, 29, 'Cu')]
```

Ćwiczenia/przykłady

- Ćwiczenie/przykład 4.2:
“Baza” osób – standardowe funkcje
- Ćwiczenie/przykład 4.3:
“Baza” osób – sortowanie



Funkcja lambda

- **Funkcja lambda** – anonimowy odpowiednik funkcji, której ciało składa się z tylko jednej instrukcji **return**
- Funkcja lambda ma postać:

Składnia funkcji lambda

```
lambda parameters: expression
```

gdzie:

<i>parameters</i>	opcjonalne parametry pozycyjne separowane przecinkami
<i>expression</i>	wyrażenie niezawierające: <ul style="list-style-type: none">– pętli (wyrażenie warunkowe jest dopuszczalne)– instrukcji return– instrukcji yield

Wyrażenia lambda

- Wartością funkcji lambda jest funkcja anonimowa
- Kiedy taka funkcja zostanie wywołana, to wynikiem tej operacji jest wynik wykonania wyrażenia *expression*
- Jeżeli wyrażenie *expression* jest krotką, to trzeba je ująć w nawiasy

Funkcja lambda

Funkcja przeliczająca stopnie Celsjusza na Fahrenheita

```
>>> def cels2fahr(c_degrees):
...     return 1.8 * c_degrees + 32
...
>>> print(cels2fahr(0))
32.0
>>> print(cels2fahr(100))
212.0
```

Funkcja anonimowa (lambda)

```
>>> cels2fahr = lambda c_degrees: 1.8 * c_degrees + 32
...
>>> print(cels2fahr(0))
32.0
>>> print(cels2fahr(100))
212.0
```

Ćwiczenia/przykłady

- Ćwiczenie/przykład 4.4:
“Baza” osób – funkcje lambda



ĆWICZENIA

Typowanie dynamiczne

- Python jest językiem dynamicznie typowanym
- Oznacza to brak konieczności deklarowania typów zmiennych i parametrów
- Typ jest wnioskowany w momencie jego użycia, w oparciu o przypisaną wartość
- Zaletą takiego podejścia jest możliwość szybszego tworzenia kodu
- Kod może być bardziej uniwersalny i elastyczny

Typowanie dynamiczne – przykład

Typowanie dynamiczne

```
>>> def get_first_name(full_name):
...     return full_name.split()[0]
...
>>> fallback_name = {'first_name': 'Nieznajomy',
...                     'last_name': 'Nieznajomy'}
>>>
>>> def greeting(full_name):
...     if full_name:
...         first_name = get_first_name(full_name)
...     else:
...         first_name = get_first_name(fallback_name)
...     print(f'Witaj {first_name}!')
...
...
```

Typowanie dynamiczne – przykład

Uruchomienie skryptu

```
>>> greeting('Jan Kowalski')
Witaj Jan!

>>> greeting(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 2, in greeting
      File "<stdin>", line 2, in get_first_name
AttributeError: 'dict' object has no attribute 'split'
```

- Wykonanie kodu z danymi nieprawidłowego typu zakończy się zgłoszeniem błędu wykonania
- Nie ma możliwości wykrycia błędu na wcześniejszym etapie – skrypt można uruchamiać wielokrotnie, a na błąd natrafić przypadkiem...

Typowanie statyczne

- Od Pythona 3.6 można stosować **typowanie statyczne**
- Umożliwia ono określenie w kodzie źródłowym oczekiwanych typów dla: zmiennych, parametrów funkcji, zwracanej wartości
- W przypadku chęci określenia typów elementów dla kontenerów można posłużyć się modelem *typing*
- Oba podejścia typowania (dynamiczne i statyczne) można łączyć ze sobą

Typowanie statyczne – przykład

Typowanie statyczne

```
>>> def get_first_name(full_name: str) -> str:  
...     return full_name.split()[0]  
...  
>>> fallback_name: dict = {'first_name': 'Nieznajomy',  
...                         'last_name': 'Nieznajomy'}  
>>>  
>>> def greeting(full_name: str) -> None:  
...     first_name: str  
...     if full_name:  
...         first_name = get_first_name(full_name)  
...     else:  
...         first_name = get_first_name(fallback_name)  
...     print(f'Witaj {first_name}!')
```

Ćwiczenia/przykłady

- Ćwiczenie/przykład 4.5:
Wieczny kalendarz – typowanie statyczne



Dokumentowanie kodu

- Jeżeli pierwszą instrukcją ciała funkcji jest literał tekstowy, to kompilator traktuje go jak literał dokumentujący
- Standardowo opis rozciąga się na wiele linii, więc literał ogranicza się potrójnymi apostrofami lub potrójnymi cudzysłowami
- Jest on związany z atrybutem funkcji o nazwie *doc*

Dokumentowanie kodu

- Zgodnie z konwencją:
 - **pierwsza linia** powinna być zwartym opisem przeznaczenia funkcji, zaczynającym się dużą literą i kończącym kropką
 - nie powinna zawierać nazwy funkcji
 - jeśli opis obejmuje wiele linii, to **druga linia** powinna być pusta
 - **kolejne linie** powinny być uformowane w akapity odseparowane pustymi liniami
 - powinny zawierać takie informacje jak: parametry, warunki wstępne, zwracana wartość, efekty uboczne
 - **na końcu** mogą się znaleźć: dalsze wyjaśnienia, odnośniki do bibliografii, przykłady użycia

Dokumentowanie kodu – przykład

Funkcja wraz z dokumentacją

```
>>> def sum_args(*numbers):
...     """Returns the sum of multiple numerical arguments.
...
...     The arguments are zero or more numbers.
...     The result is their sum.
...
...     """
...     return sum(numbers)
```

Opis funkcji

```
>>> print(sum_args.__doc__)
Returns the sum of multiple numerical arguments.
```

The arguments are zero or more numbers.
The result is their sum.

```
>>>
```

Plan szkolenia

- 1 WPROWADZENIE DO JĘZYKA PYTHON
- 2 PODSTAWOWE KONCEPCJE
- 3 ZŁOŻONE TYPY DANYCH
- 4 PROGRAMOWANIE FUNKCYJNE
- 5 **KLASY I OBIEKTY**
- 6 MODUŁY I PAKIETY
- 7 OPERACJE NA PLIKACH
- 8 WYJĄTKI
- 9 WAŻNE WBUDOWANE MODUŁY I BIBLIOTEKI

Plan modułu

5 KLASY I OBIEKTY

- paradygmat OOP (Object-Oriented Programming)
- klasy i obiekty – podstawy
- atrybuty klasy
- metody
- instancje klasy
- atrybuty instancji
- kontrola dostępu do atrybutów
- definiowanie i wykorzystanie właściwości (properties)
- dziedziczenie
- porządek poszukiwania atrybutów
- metody magiczne



Programowanie zorientowane obiektowo

- Python jest **językiem zorientowanym obiektowo** (*OOL – Object-Oriented Language*)
- Nie wymusza stosowania tego paradygmatu na wyłączność
- Wspiera także **programowanie proceduralne** oparte na modułach i funkcjach
- Można samodzielnie wybrać właściwy paradygmat dla każdej części programu

Klasy i instancje

- Podobnie jak w innych językach zorientowanych obiektowo, **klasa** (*class*) jest właściwie definicją **typu danych** (*data type*) – wszystkie wbudowane typy danych w Pythonie są klasami
- Klasy można **instancjonować** (*instantiate*) w celu utworzenia **instancji**, czyli obiektów tego typu
- W Pythonie te idee realizuje się z pomocą **obiektów klasy** i **obiektów instancji** (*class and instance objects*)

Programowanie OOP

- Paradygmat programowania zorientowanego obiektowo (OOP) umożliwia zgrupowanie w pojedynczych jednostkach funkcjonalnych, zw. **klasami**:

stanu (state) | czyli danych

zachowania (behavior) | czyli kodu operacji na danych

- Klasę można traktować jak **szablon** (prototyp) w oparciu o który tworzone są instancje
- Klasa definiuje **zbior atrybutow** charakteryzujących wszystkie instancje klasy

Klasy w Pythonie

- Klasa jest **obiektem Pythona**
- Jest pełnoprawnym obiektem, z wszystkimi możliwościami, jakimi dysponują inne obiekty – jest tzw. **obiektem pierwszej kategorii** (*first-class object, first-class citizen*)
- Z tego powodu:
 - klasę można przekazać jako argument w wywołaniu funkcji
 - funkcja może w wyniku wywołania zwrócić obiekt klasy
 - klasa może zostać związana ze zmienną (lokalną bądź globalną)
 - klasa może być elementem kontenera lub pełnić rolę klucza w słowniku
 - klasa może być atrybutem obiektu

Ogólna definicja klasy

Składnia definicji klasy

```
class ClassName (base-classes) :  
    statement (s)
```

<i>ClassName</i>	identyfikator reprezentujący nazwę klasy
<i>base-classes</i>	opcjonalna krotka obiektów klas, reprezentujących nadklasy w razie pominięcia nadklas, można także opuścić nawiasy (☞ więcej nt. dziedziczenia w dalszej części rozdziału...)
<i>statements</i>	niepusta sekwencja instrukcji, stanowiących ciało klasy

Nazwa klasy

- **Nazwa klasy** musi być poprawnym identyfikatorem
- Zgodnie z umową, w stosunku do klas stosujemy konwencję nazewniczą *CamelCase*
- Najprostsza klasa ma postać:

Definicja klasy – przykład

```
>>> class EmptyClass: pass  
...
```

Atrybuty obiektów klas

Definicja klasy z atrybutami – przykład

```
>>> class A:  
...     x = 123      # atrybut klasy  
...     y = x + 1    # atrybut klasy  
...  
>>> print(A.x, A.y)  
123 124
```

- Instrukcje w ciele klasy zawierające odwołania do atrybutów klasy wykorzystują **nazwy proste** (niekwalifikowane)
- Na zewnątrz klasy do odwołania się do jej atrybutu wykorzystuje się **nazwy kwalifikowane**

Atrybuty obiektów klas

- Atrybuty obiektu klasy można także tworzyć i zmieniać poza ciałem klasy:

Definicja klasy

```
>>> class EmptyClass: pass  
...  
>>> EmptyClass.x = 123 # atrybut  
>>> print(EmptyClass.x)  
123
```

- Tak utworzony atrybut nie różni się niczym od atrybutu utworzonego w ciele klasy

Niejawne atrybuty obiektów klas

- Instrukcja definicji klasy (instrukcja `class`) tworzy obiekt klasy i ustawia szereg **niejawnych atrybutów** klasy
- Wśród nich są:

`__name__`

nazwa klasy

`__bases__`

krotka obiektów rozszerzanych klas

`__doc__`

opis klasy

`__dict__`

mapa służąca klasie jako lokalna przestrzeń nazw

`__module__`

nazwa modułu w którym zdefiniowana jest klasa

w trybie interaktywnym ten atrybut ma wartość `__main__`

Odczyt atrybutu klasy

- Odczyt atrybutu klasy, gdy nie jest on deskryptorem, przebiega następująco:

Odczyt atrybutu klasy

```
>>> class A:  
...     x = 123  
...  
>>> # odczyt atrybutu klasy  
... print(A.x)  
123  
  
>>> # równoważnie:  
... print(A.__dict__['x'])  
123
```

- Atrybuty klasy mogą odnosić się także do **obiektów funkcji** (instrukcja `def`)
- Atrybuty klasy związane z funkcjami nazywa się **metodami** (*methods, callable attributes*)
- Atrybuty klasy są **współdzielone** przez wszystkie instancje klasy

Metody instancyjne

- Metody, które chcą mieć dostęp do atrybutów danej instancji muszą deklarować w liście parametrów, na pierwszej pozycji, **obowiązkowy parametr**, odnoszący się do **bieżącej instancji**, tzn. tej na której metoda jest wywoływana
- Zgodnie z konwencją temu parametrowi nadaje się nazwę `self`
- Takie metody nazywamy **metodami instancyjnymi** (*instance methods*)

Metody instancyjne

Definicja klasy z metodą instancyjną – przykład

```
>>> class MyClass:  
...     # metoda instancyjna  
...     def hello(self):  
...         print('Hello')  
... 
```

- Poprzez parametr `self` metody instancyjne mają swobodny dostęp do atrybutów i innych metod tego samego obiektu
- Metody instancyjne woła się na rzecz instancji

Instancje klasy

- Aby utworzyć **instancję klasy** (*class instance*), należy wywołać **obiekt klasy**, tak, jakby była to funkcja (funkcja konstruktora)
- Każde takie wywołanie zwraca nową instancję typu klasy
- Ten proces nazywamy **instancjonowaniem** (*instantiation*)

Tworzenie instancji klasy *MyClass*

```
>>> mc = MyClass()      # tworzenie instancji
>>> mc.hello()         # wywołanie metody instancyjnej
Hello
>>>
```

Instancje klasy

- Wbudowana funkcja `isinstance` może posłużyć do weryfikacji, **czy instancja jest danego typu** lub typu podklasy

Test typu instancji

```
>>> isinstance(mc, MyClass) # test typu instancji  
True
```

Metoda specjalna `__new__`

- Każda klasa posiada własną metodę specjalną `__new__` lub ją dziedziczy

Składnia metody `__new__`

```
__new__(cls[, ...])
```

- Metoda służy do **tworzenia nowych, niezainicjowanych instancji klasy**
- Pierwszy z parametrów `cls` jest obiektem klasy, której instancja jest tworzona
- Opcjonalne argumenty przekazane metodzie `__new__` są przez nią ignorowane i przekazywane metodzie specjalnej `__init__`

Metoda specjalna `__init__`

- Podczas tworzenia instancji, Python **niejawnie wywołuje specjalną metodę `__init__`** ("dunder init")

Składnia metody `__init__`

```
__init__(self[, args...])
```

- Można tę metodę zdefiniować samodzielnie (nadpisać) lub ją dziedziczyć

Tworzenie instancji

- Przykładowa instrukcja utworzenia instancji klasy A:

Tworzenie instancji

```
>>> a = A(123)
```

jest równoważna:

Tworzenie instancji

```
>>> a = A.__new__(A, 123)
>>> if isinstance(a, A):
...     type(a).__init__(a, 123)
...
...
```

Metoda specjalna `__init__`

- Metoda `__init__` umożliwia dokonanie wszelkiej niezbędnej **inicjalizacji instancji** (najczęściej utworzenia i zainicjowania atrybutów)
- Argumenty podawane podczas tworzenia instancji są przekazywane metodzie `__init__`
- Metoda `__init__` nie może zwrócić żadnej innej wartości niż `None`
– w przeciwnym razie zostanie zgłoszony wyjątek `TypeError`

Metoda specjalna `__init__` – przykład

- Po utworzeniu instancji można uzyskać dostęp do jej atrybutów (danych i metod):

Tworzenie instancji klasy

```
>>> class A:  
...     def __init__(self, n):  
...         print('utworzono instancję klasy A')  
...         self.x = n  
...  
>>> a = A(123)  
utworzono instancję klasy A  
  
>>> print(a.x)      # atrybut instancyjny  
123
```

Atrybuty instancji

- Atrybuty instancji można także dodać do już istniejącej instancji

Dodawanie atrybutów

```
>>> class A: pass  
...  
>>> a = A()  
>>> a.x = 123  
>>> print(a.x)  
123
```

Atrybuty instancji

- Utworzenie instancji **niejawnie ustawia** dwa **atrybuty instancji**:

`__class__` | obiekt klasy, do którego należy dana instancja

`__dict__` | słownik do przechowywania pozostałych atrybutów instancji

Atrybuty instancji

```
>>> print(a.__class__.__name__)
```

A

```
>>> print(a.__dict__)
```

{'x': 123}

Odczyt atrybutu instancji

```
>>> class A:  
...     def __init__(self, value):  
...         self.x = value    # atrybut instancji  
...  
>>> a = A(123)  
  
>>> # odczyt atrybutu  
>>> print(a.x)  
123  
  
>>> # równoważnie:  
>>> print(a.__dict__['x'])  
123
```

Dostęp do atrybutów

- Do **dostępu do atrybutów** można wykorzystać operator kropki
- Równoważnie można stosować funkcje:

Funkcje dostępu do atrybutów

```
getattr(obj, name[, default])
setattr(obj, name, value)
delattr(obj, name)

hasattr(obj, name)
```

- Parametr *obj* może wskazywać **instancję** lub **obiekt klasy**

Ćwiczenia/przykłady

- Ćwiczenie/przykład 5.1:
Definiowanie klas i obiektów



ĆWICZENIA

Prywatne identyfikatory

- Identyfikatory rozpoczynające się pojedynczym znakiem podkreślenia `_` służą do definiowania **prywatnych** zmiennych, funkcji, metod i klas
- Niektóre instrukcje importu pomijają takie identyfikatory → prywatność na poziomie modułu
- **Zgodnie z konwencją**, są one prywatne względem zasięgu w którym zostały zdefiniowane

Prywatne identyfikatory

- Nie jest to realizacja “prawdziwej” prywatności – elementy są nadal dostępne w sposób bezpośredni z innych modułów
- Kompilator **nie wymusza i nie pilnuje w żaden sposób realizacji prywatności** – jest to **tylko umowa** respektowana przez programistów
- Rolę znaku podkreślenia dość dobrze oddaje termin “*weak internal-use indicator*”

Silnie prywatne identyfikatory

- Identyfikatory rozpoczynające się, ale nie kończące, dwoma znakami podkreślenia _ są **silnie prywatne**
- Kompilator Pythona niejawnie przekształca takie nazwy zgodnie z regułą:
_identifier → _ClassName__identifier
- Zmniejsza to ryzyko przypadkowego zdublowania nazw atrybutów, metod, czy zmiennych globalnych
- Ma to szczególne zastosowanie w podklasach

Silnie prywatne identyfikatory – przykład

Dostęp do atrybutów

```
>>> class A:  
...     x = 111  
...     _y = 222    # atrybut prywatny  
...     __z = 333   # atrybut silnie prywatny  
...  
>>> print(A.x)  
111  
>>> print(A._y)  
222  
>>> print(A.__z)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
AttributeError: type object 'A' has no attribute '__z'  
>>> print(A._A__z)  
333
```

Hermetyzacja danych

- W wielu sytuacjach stajemy przed problemem **kontroli dostępu do atrybutów**
- W językach zorientowanych obiektowo można ten problem rozwiązać za pomocą **hermetyzacji (encapsulation)**
- Hermetyzacja polega na zablokowaniu bezpośredniego dostępu do danych obiektu z zewnątrz – dane obiektu należy zadeklarować jako prywatne
- Modyfikacja danych z zewnątrz jest możliwa tylko poprzez wywołanie **metod dostępowych**
- Przyjęło się nazywać metody służące do ustawiania wartości – **setterami**, a do odczytu danych – **getterami**

Hermetyzacja

- Trywialna realizacja tego pomysłu mogłaby wyglądać następująco:

Hermetyzacja

```
>>> class A:  
...     def __init__(self, value):  
...         self.__x = value  
...     def get_x(self):  
...         return self.__x  
...     def set_x(self, value):  
...         self.__x = value  
...     ...
```

Ćwiczenia/przykłady

- Ćwiczenie/przykład 5.2:
Hermetyzacja



ĆWICZENIA

Hermetyzacja

- Takie rozwiązanie jest mało eleganckie – dostęp do danych jest skomplikowany i mało czytelny (zwł. modyfikacja danych)
- Usunięcie metod dostępowych i uczynienie atrybutu `x` publicznym, może uprościć kod, ale... to rozwiązanie **nie umożliwia kontroli nad dopuszczalnymi wartościami atrybutu**
- Taki walidujący kod można zatrzymać tylko w metodzie (najlepiej w setterze)...

Hermetryzacja

Dostęp do atrybutu

```
>>> class A:  
...     def __init__(self, value):  
...         self.x = value  
...     def get_x(self):  
...         return self.x  
...     def set_x(self, value):  
...         if value < 0:  
...             self.x = 0  
...         elif value > 9:  
...             self.x = 9  
...         else:  
...             self.x = value  
... 
```

Wykorzystanie właściwości

- Niestety, ten kod nadal nie jest poprawny
- Klasa nie jest shermetyzowana i dostęp do atrybutu `x` jest możliwy zarówno poprzez getter i setter, jak i w sposób bezpośredni
- Rozwiązaniem tego problemu jest zdefiniowanie atrybutu `x` jako **właściwości (property)**
- Właściwości umożliwiają dostęp bezpośredni do wartości, ale faktycznie, w zależności od kontekstu, wywoływana jest odpowiednia metoda dostępową
- Jednym ze sposobów utworzenia właściwości jest zastosowanie funkcji *property*

Definiowanie właściwości

Definiowanie właściwości

```
>>> class A:  
...     def __init__(self, value):  
...         self.__set_x(value)  
...     def __get_x(self):  
...         return self.__x  
...     def __set_x(self, value):  
...         if value < 0:  
...             self.__x = 0  
...         elif value > 9:  
...             self.__x = 9  
...         else:  
...             self.__x = value  
...     x = property(__get_x, __set_x)  
... 
```

Użycie właściwości

Użycie właściwości

```
>>> a = A(6)      # setter  
>>> print(a.x)  # getter  
6  
>>> a.x *= 2    # getter + setter  
>>> print(a.x)  # getter  
9  
>>> a.x = 5     # setter  
>>> print(a.x)  # getter  
5  
>>> a.x = -8    # setter  
>>> print(a.x)  # getter  
0
```

Ćwiczenia/przykłady

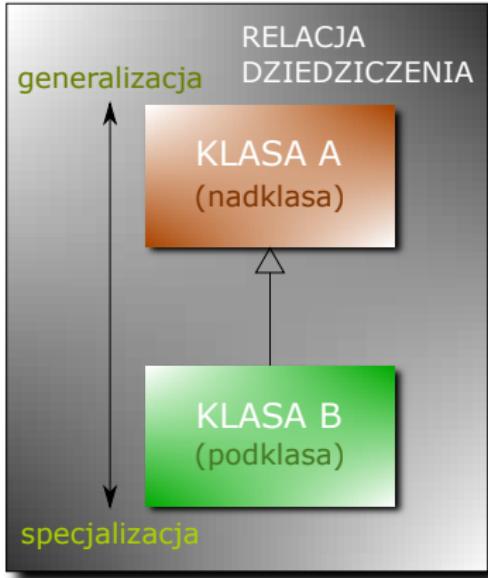
- Ćwiczenie/przykład 5.3:
Wykorzystanie właściwości



ĆWICZENIA

Relacja dziedziczenia

- Jedną z zalet podejścia obiektowego jest możliwość **specjalizacji klasy** (*class specialization*), czyli wyprowadzenia nowej klasy na bazie klasy istniejącej
- Nowa klasa **dziedziczy** wszystkie **atrybuty** (dane i metody) z klasy bazowej
- Klasę bazową nazywamy **nadklassą** (*superclass*), zaś klasę potomną – **podklassą** (*subclass*)



Rodzaje dziedziczenia

- Gdy klasa może posiadać tylko jednego przodka, to mówimy o **dziedziczeniu jednobazowym** (*single inheritance*)
- Jedynym wyjątkiem od tej reguły jest klasa będąca przodkiem wszystkich innych klas – ta, jako jedyna nie posiada swojego “rodzica”
- Taka struktura zależności prowadzi do drzewa hierarchii dziedziczenia
- Jeśli dopuścimy możliwość posiadania przez klasę wielu przodków, to mamy do czynienia z **dziedziczeniem wielobazowym** (*multiple inheritance*)

Relacja dziedziczenia

- Klasa potomna może **dodać nowe lub przedefiniować istniejące atrybuty**
- Można **dziedziczyć**, tzn. **rozszerzać** dowolne klasy: wbudowane, ze standardowych bibliotek oraz własne
- Dziedziczenie ułatwia tworzenie kodu, gdyż można wykorzystać istniejącą i przetestowaną funkcjonalność jako bazę dla nowych klas
- Instancje klas potomnych można przekazywać do funkcji i metod napisanych dla oryginalnej klasy → polimorfizm

Dziedziczenie w Pythonie

- Dziedziczenie jest mechanizmem **współdzielenia funkcjonalności** pomiędzy klasami
- W Pythonie mamy do czynienia z **dziedziczeniem wielobazowym** – klasy mogą posiadać wielu przodków
- Definiując klasę, za jej nazwą można wyliczyć wszystkich jej bezpośrednich przodków (rodziców)
- Zawartość listy z klasami bazowymi jest dostępna poprzez atrybut `_bases_`

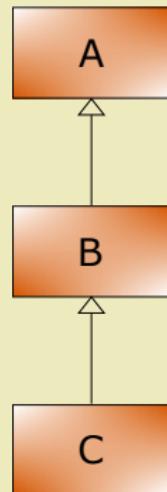
Relacja dziedziczenia

- Relacja dziedziczenia jest relacją **przechodnią** – jeśli klasa *C* rozszerza klasę *B*, a klasa *B* rozszerza klasę *A*, to klasa *C* rozszerza również klasę *A*
- Do testowania relacji dziedziczenia można wykorzystać wbudowaną funkcję *issubclass*

Relacja dziedziczenia – przykład

Relacja dziedziczenia

```
>>> class A: pass  
...  
>>> class B(A): pass  
...  
>>> class C(B): pass  
...  
>>> print(issubclass(C, C))  
True  
>>> print(issubclass(C, B))  
True  
>>> print(issubclass(C, A))  
True  
>>> print(issubclass(B, A))  
True  
>>> print(issubclass(A, B))  
False
```



Dziedziczenie w Pythonie

- Klasy bazowe mogą mieć także swoich rodziców, itd.
- Pominiecie w definicji klasy jej klas bazowych oznacza, że tworzona klasa **niejawnie rozszerza** klasę *object*
- Klasa *object* jest **nadklasą wszystkich klas** Pythona

Klasy bazowe

```
>>> class A(object): pass  
...  
>>> class B: pass  
...  
>>> print(A.__bases__, B.__bases__)  
(<class 'object'>,) (<class 'object'>,)
```

Dziedziczenie

- Klasa dziedzicząca po innej klasie uzyskuje dostęp do atrybutów klasy przodka

Dziedziczenie

```
>>> class A:  
...     x = 1  
...     def f(self): return 'f() called from A'  
...  
>>> class B(A):  
...     y = 2  
...     def g(self): return 'g() called from B'  
...  
>>> print(B.x, B.y)  
1 2  
>>> b = B()  
>>> print(b.f())  
f() called from A  
>>> print(b.g())  
g() called from B
```

Identyfikacja atrybutów

- Gdy w programie wystąpi odwołanie do atrybutu, w celu jego identyfikacji podejmowanych jest w kolejności kilka kroków:
 - najpierw atrybut jest poszukiwany w słowniku *dict*
 - jeśli nie zostanie tam odnaleziony, to poszukiwanie rozciąga się na wszystkie klasy wymienione w atrybucie *bases* w **określonej kolejności**
 - ponieważ klasy bazowe również mogą posiadać swoich przodków, to proces poszukiwania niejawnie dotyczy wszystkich przodków (niezależnie od pokolenia)
 - poszukiwanie kończy się w momencie znalezienia atrybutu o podanej nazwie

Kolejność poszukiwania – MRO

- Porządek przeszukiwania dotyczy **wszystkich typów atrybutów**, choć historycznie stosuje dla niego termin *MRO* (*method resolution order*)
- Poszukiwanie atrybutu o podanej nazwie odbywa się poprzez przeglądanie klas bezpośrednich przodków w kolejności **od lewej do prawej**
- Przed przejściem do kolejnej klasy na tym samym poziomie następuje analiza **w głąb** (zgodnie z relacją dziedziczenia – w kierunku klasy *object*)

Kolejność poszukiwania – MRO

- Każda klasa i wbudowany typ posiada atrybut tylko-do-odczytu `__mro__`, którego wartością jest krotka przeszukiwanych, w odpowiedniej kolejności klas
- Można także użyć wbudowanej, bezargumentowej funkcji `mro`
- Można ją wywołać na rzecz obiektu klasy
- Metoda jest wywoływana przy instancjonowaniu klasy, a jej wynik zapamiętywany w atrybucie `__mro__`

Nadpisywanie atrybutów

- Poszukiwanie atrybutu odbywa się zgodnie z porządkiem MRO (typowo w górę drzewa hierarchii dziedziczenia) i kończy w momencie jego odnalezienia
- Klasy potomne przeszukiwane są zawsze wcześniej – przed klasami przodków
- W konsekwencji, jeśli podkلاśa zdefiniuje atrybut o identycznej nazwie co nadklaśa, to zostanie znaleziona definicja w podklaśsie i poszukiwanie tu się zakończy
- Taki mechanizm jest nazywany **przedefiniowywaniem** lub **nadpisywaniem** (*override*) atrybutów

Nadpisywanie atrybutów – przykład

Nadpisywanie atrybutów

```
>>> class A:  
...     a = 111  
...     b = 222  
...     def f(self): print('metoda f() z klasy A')  
...     def g(self): print('metoda g() z klasy A')  
...  
>>> class B(A):  
...     b = 333 # nadpisanie  
...     c = 444  
...     d = 555  
...     def g(self): print('metoda g() z klasy B') # nadpisanie  
...     def h(self): print('metoda h() z klasy B')  
...
```

Nadpisywanie atrybutów – przykład cd.

Nadpisywanie atrybutów

```
>>> print(B.a, B.b, B.c, B.d)  
111 333 444 555
```

```
>>> b = B()
```

```
>>> b.f()  
metoda f() z klasy A
```

```
>>> b.g()  
metoda g() z klasy B
```

```
>>> b.h()  
metoda h() z klasy B
```

Delegacja do metod nadklasy

- Czasami przeddefiniowująca metoda w podklasie może chcieć wykorzystać zachowanie oryginalnej metody nadpisywanej
- W takiej sytuacji można wykorzystać **delegację** (*delegation*)
- Metoda może odwołać się do obiektu metody z klasy bazowej i przekazać jej potrzebne argumenty (włącznie z *self*)

Delegacja do metod nadklasy – przykład

Delegacja do metod nadklasy

```
>>> class A:  
...     def greet(self, name): print('Welcome', name)  
...  
>>> class B(A):  
...     def greet(self, name):  
...         A.greet(self, name)    # delegacja  
...         print('It\'s nice to see you')  
...  
>>> b = B()  
>>> b.greet('John')  
Welcome John  
It's nice to see you
```

Inicjalizacja podklasy

- Jeśli klasa potomna przedefiniuje metodę `__init__`, to zostanie ona wywołana podczas tworzenia instancji tej klasy
- Metoda automatycznie nie wywoła przedefiniowanej metody z nadklasy, w efekcie czego instancja może nie być do końca poprawnie zainicjalizowana
- Wywołanie metody `__init__` z nadklasy jest **odpowiedzialnością programisty**

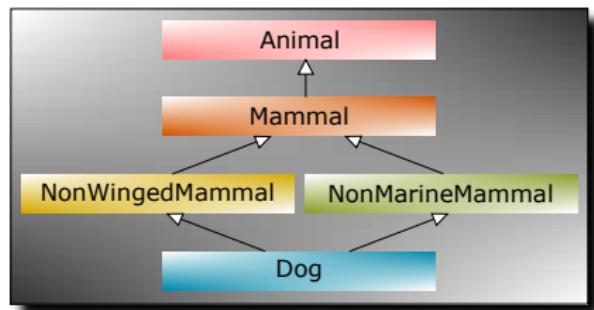
Inicjalizacja podklasy – przykład

Przedefiniowanie metody `__init__`

```
>>> class Point2D:  
...     def __init__(self, x_coord, y_coord):  
...         self.x = x_coord  
...         self.y = y_coord  
...  
>>> class Point3D(Point2D):  
...     def __init__(self, x_coord, y_coord, z_coord):  
...         Point2D.__init__(self, x_coord, y_coord)  
...         self.z = z_coord  
...  
>>> p = Point3D(1, 2, 3)  
>>> print(p.x, p.y, p.z)  
1 2 3
```

Nadpisywanie metod w dziedziczeniu wielobazowym

- Przedstawiony sposób delegacji będzie działał zarówno w Pythonie 3.x, jak i 2.x
- Problematyczne może być jego zastosowanie w przypadku dziedziczenia wielobazowego, gdy grafy mają kształt diamentu (*diamond-shaped graphs*)
- Może to prowadzić do wielokrotnego wywołania tej samej metody



Nadpisywanie metod w dziedziczeniu wielobazowym

- Rozwiązaniem tego problemu może być użycie wbudowanego typu *super*
- Wywołanie *super()* zwraca obiekt proxy, który umożliwia wywoływanie metod klasy rodzica poprzez delegację
- W Pythonie 2.x wywołanie musiało mieć postać:
super(subclass, obj)
- W Pythonie 3.x wywołanie może być bezargumentowe

Ćwiczenia/przykłady

- Ćwiczenie/przykład 5.4:
Dziedziczenie – pracownicy i kierownicy zespołów



Metody klasy *object*

- Do sprawdzenia jakie metody są dostępne w klasie *object* można wykorzystać poniższe wyrażenie listowe:

Metody klasy *object*

```
>>> m = [method_name for method_name in dir(object)
...                     if callable(getattr(object, method_name))]
>>> print(m)
['__class__', '__delattr__', '__dir__', '__eq__', '__format__',
 '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__le__', '__lt__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__']
```

Metody specjalne

- Klasa może dziedziczyć lub przeddefiniować metody specjalne (*dunder methods, magic methods*)
- Każda metoda specjalna jest związana z jakąś specyficzna operacją
- Python niejawnie wywołuje te metody podczas wykonywania operacji
- W większości przypadków wartość zwracana przez metodę specjalną jest wynikiem operacji – próba wykonania operacji, gdy brak odpowiedniej metody specjalnej skutkuje zgłoszeniem wyjątku
- Klasa, która definiuje lub dziedziczy te metody, umożliwia swoim instancjom kontrolę powiązanych operacji

Podstawowe metody specjalne

METODA SPECJALNA	SPOSÓB UŻYCIA	ZNACZENIE
<code>__bool__(self)</code>	<code>bool(x)</code>	zwraca wartość logiczną dla x
<code>__format__(self, fmt_spec)</code>	<code>"{0}".format(x)</code>	umożliwia użycie <code>str.format()</code> w stosunku do własnych klas
<code>__hash__(self)</code>	<code>hash(x)</code>	umożliwia, aby x był kluczem słownika lub elementem zbioru
<code>__init__(self, args)</code>	<code>x = X(args)</code>	wywoływana podczas inicjalizacji obiektu
<code>__new__(cls, args)</code>	<code>x = X(args)</code>	wywoływana podczas tworzenia obiektu
<code>__del__(self)</code>	<code>del x</code>	może być wywołana (nie ma takiej gwarancji), gdy liczba referencji do obiektu spadnie do 0
<code>__repr__(self)</code>	<code>repr(x)</code>	zwraca reprezentację tekstową x taką, aby w miarę możliwości <code>eval(repr(x)) == x</code>
<code>__repr__(self)</code>	<code>ascii(x)</code>	zwraca reprezentację tekstową x zawierającą jedynie znaki ASCII
<code>__str__(self)</code>	<code>str(x)</code>	zwraca reprezentację tekstową x przeznaczoną dla człowieka

Metody specjalne związane z porównywaniem

METODA SPECJALNA	SPOSÓB UŻYCIA	ZNACZENIE
<code>__lt__(self, other)</code>	$x < y$	zwraca <i>True</i> , jeśli x jest mniejsze niż y
<code>__le__(self, other)</code>	$x \leq y$	zwraca <i>True</i> , jeśli x jest mniejsze lub równe y
<code>__eq__(self, other)</code>	$x == y$	zwraca <i>True</i> , jeśli x jest równe y
<code>__ne__(self, other)</code>	$x != y$	zwraca <i>True</i> , jeśli x nie jest równe y
<code>__gt__(self, other)</code>	$x > y$	zwraca <i>True</i> , jeśli x jest większe niż y
<code>__ge__(self, other)</code>	$x \geq y$	zwraca <i>True</i> , jeśli x jest większe lub równe y

Zachowanie nowych klas użytkownika

- Standardowo instancje własnych klas użytkownika **wspierają operator porównania** `==`
- Jeśli klasa nie przedefiniuje metody `__eq__` (aby zaimplementować operator `==`), to odziedziczy ją z klasy `object`
- Wtedy standardowo instancje będą porównywane na podstawie ich tożsamości (funkcja `id`)
- Z tego powodu wynikiem porównania tych obiektów zawsze będzie **False**

Zachowanie nowych klas użytkownika

- Metoda skrótu `__hash__` zwraca wartość całkowitą opartą na wartości obiektu
- Ponieważ standardowo wartością obiektu jest jego tożsamość, to z tego powodu wszystkie instancje (które nie nadpiszą metody `__hash__`) są:
 - **niemutowalne** (wartość obiektu nie ulega zmianie w całym cyklu życia)
 - **haszowalne**
- Takie obiekty mogą być elementami zbiorów oraz kluczami słowników

Zachowanie nowych klas użytkownika

Test instancji własnych klas

```
>>> class A:  
...     def __init__(self, value):  
...         self.x = value  
...  
>>> a1 = A(1)  
>>> a2 = A(1)  
>>> print(a1 == a2)      # porównanie instancji  
False  
  
>>> s = {a1, a2}          # instancje są haszowalne  
>>> print(hash(a1))  
154757017969  
>>> print(hash(a2))  
154757017899
```

Zachowanie nowych klas użytkownika

- Przedefiniowanie w klasie metody specjalnej `__eq__` powoduje, że instancje **nie są haszowalne**
- Python zablokuje standardową implementację funkcji skrótu, gdyż zmieniła się definicja wartości obiektu
- Aby instancje były haszowalne, trzeba dodatkowo przedefiniować metodę `__hash__`

Zachowanie nowych klas użytkownika

Test instancji własnych klas

```
>>> class A:  
...     def __init__(self, value):  
...         self.x = value  
...     def __eq__(self, other):  
...         return (self.__class__ == other.__class__  
...                 and self.x == other.x)  
...     def __hash__(self):  
...         return hash(self.x)  
...  
>>> a1 = A(1)  
>>> a2 = A(1)  
>>> print(a1 == a2)      # porównanie instancji  
True  
>>> s = {a1, a2}          # instancje są haszowalne
```

Numeryczne metody specjalne

METODA SPECJALNA	SPOSÓB UŻYCIA	ZNACZENIE
<code>__abs__(self)</code>	<code>abs(x)</code>	wartość bezwzględna
<code>__int__(self)</code>	<code>int(x)</code>	konwersja do typu <i>int</i>
<code>__float__(self)</code>	<code>float(x)</code>	konwersja do typu <i>float</i>
<code>__complex__(self)</code>	<code>complex(x)</code>	konwersja do typu <i>complex</i>
<code>__index__(self)</code>	<code>bin(x), oct(x), hex(x)</code>	zwraca wartość całkowitą
<code>__round__(self, digits)</code>	<code>round(x, digits)</code>	zaokrąglenie
<code>__pos__(self)</code>	<code>+x</code>	działanie jednoargumentowego operatora +
<code>__neg__(self)</code>	<code>-x</code>	działanie jednoargumentowego operatora -
<code>__add__(self, other)</code> <code>__radd__(self, other)</code> <code>__iadd__(self, other)</code>	$x + y$ $y + x$ $x += y$	operacja dodawania
<code>__sub__(self, other)</code> <code>__rsub__(self, other)</code> <code>__isub__(self, other)</code>	$x - y$ $y - x$ $x -= y$	operacja odejmowania
<code>__mul__(self, other)</code> <code>__rmul__(self, other)</code> <code>__imul__(self, other)</code>	$x * y$ $y * x$ $x *= y$	operacja mnożenia
<code>__mod__(self, other)</code> <code>__rmod__(self, other)</code> <code>__imod__(self, other)</code>	$x \% y$ $y \% x$ $x \%= y$	operacja modulo

Numeryczne metody specjalne

METODA SPECJALNA	SPOSÓB UŻYCIA	ZNACZENIE
<code>__truediv__(self, other)</code> <code>__rtruediv__(self, other)</code> <code>__itruediv__(self, other)</code>	x / y y / x $x /= y$	operacja dzielenia
<code>__floordiv__(self, other)</code> <code>__rfloordiv__(self, other)</code> <code>__ifloordiv__(self, other)</code>	$x // y$ $y // x$ $x //= y$	operacja dzielenia całkowitego
<code>__divmod__(self, other)</code> <code>__rdivmod__(self, other)</code>	$divmod(x, y)$ $divmod(y, x)$	operacja <i>divmod</i>
<code>__pow__(self, other)</code> <code>__rpow__(self, other)</code> <code>__ipow__(self, other)</code>	$x ** y$ $y ** x$ $x **= y$	operacja potęgowania

Bitowe metody specjalne

METODA SPECJALNA	SPOSÓB UŻYCIA	ZNACZENIE
<code>__and__(self, other)</code> <code>__rand__(self, other)</code> <code>__iand__(self, other)</code>	$x \& y$ $y \& x$ $x \&= y$	operacja "bitowe i"
<code>__or__(self, other)</code> <code>__ror__(self, other)</code> <code>__ior__(self, other)</code>	$x y$ $y x$ $x = y$	operacja "bitowe lub"
<code>__xor__(self, other)</code> <code>__rxor__(self, other)</code> <code>__ixor__(self, other)</code>	$x ^ y$ $y ^ x$ $x ^= y$	operacja "bitowe albo"
<code>__lshift__(self, other)</code> <code>__rlshift__(self, other)</code> <code>__ilshift__(self, other)</code>	$x << y$ $y << x$ $x <<= y$	przesunięcie bitowe w lewo
<code>__rshift__(self, other)</code> <code>__rshift__(self, other)</code> <code>__rshift__(self, other)</code>	$x >> y$ $y >> x$ $x >>= y$	przesunięcie bitowe w prawo
<code>__invert__(self, other)</code>	$\sim x$	odwrócenie bitów

Metody specjalne sekwencji

METODA SPECJALNA	SPOSÓB UŻYCIA	ZNACZENIE
<code>__contains__(self, x)</code>	<code>x in y</code>	zwraca <i>True</i> , jeśli <i>x</i> jest elementem sekwencji <i>y</i> lub kluczem słownika <i>y</i>
<code>__delitem__(self, k)</code>	<code>del y[k]</code>	usuwa <i>k</i> -ty element sekwencji <i>y</i> lub element spod klucza <i>k</i> słownika <i>y</i>
<code>__getitem__(self, k)</code>	<code>y[k]</code>	zwraca <i>k</i> -ty element sekwencji <i>y</i> lub wartość spod klucza <i>k</i> słownika <i>y</i>
<code>__iter__(self)</code>	<code>for x in y: pass</code>	zwraca iterator dla elementów sekwencji <i>y</i> lub kluczy słownika <i>y</i>
<code>__len__(self)</code>	<code>len(y)</code>	zwraca ilość elementów w <i>y</i>
<code>__reversed__(self)</code>	<code>reversed(y)</code>	zwraca odwrotny iterator dla elementów sekwencji <i>y</i> lub kluczy słownika <i>y</i>
<code>__setitem__(self, k, v)</code>	<code>y[k] = v</code>	ustawia wartość <i>k</i> -tego elementu sekwencji <i>y</i> lub wartość dla klucza <i>k</i> słownika <i>y</i> na <i>v</i>

Przeciążanie operatorów

- Klasy, których instancje nie są wartościami numerycznymi, mogą przedefiniować metody specjalne związane z operatorami
- W takiej sytuacji można użyć tych operatorów w stosunku do instancji
- Nazywa się to **przeciążaniem operatorów** (*operator overloading*)
- Python nie umożliwia przeciążania metod

Ćwiczenia/przykłady

- Ćwiczenie/przykład 5.5:
Przeciążanie operatorów



Plan szkolenia

- 1 WPROWADZENIE DO JĘZYKA PYTHON
- 2 PODSTAWOWE KONCEPCJE
- 3 ZŁOŻONE TYPY DANYCH
- 4 PROGRAMOWANIE FUNKCYJNE
- 5 KLASY I OBIEKTY
- 6 MODUŁY I PAKIETY**
- 7 OPERACJE NA PLIKACH
- 8 WYJĄTKI
- 9 WAŻNE WBUDOWANE MODUŁY I BIBLIOTEKI

6 MODUŁY I PAKIETY

- obiekty modułów
- instrukcja importu
- atrybuty modułów
- instrukcja `from`
- program główny
- pakiety
- atrybuty pakietów



Moduły i pakiety

- Typowy program w Pythonie składa się z wielu **plików źródłowych**
- Każdy plik źródłowy stanowi osobny **moduł** (*module*) grupujący kod i dane
- Moduły są zwykle od siebie niezależne (pełnią rolę skryptów), przez co mogą być używane przez wiele programów
- Pełnią bardzo ważną rolę organizacyjną
- Powiązane ze sobą logicznie moduły można organizować w **pakiety** (*packages*) – struktury hierarchiczne o postaci drzewa

Obiekt modułu

- Moduł w Pythonie jest **obiektem** i zachowuje się tak, jak inne obiekty
- Można z nim związać dowolne atrybuty
- Moduł jest obiektem pierwszej kategorii i można go przypisać do zmiennej, przekazać jako argument, zwrócić z funkcji, itd.
- Moduł zwykle umieszczany jest w pliku o nazwie *nazwa-modułu.py*

Instrukcja importu

- Dzięki wykonaniu instrukcji importu można użyć dowolnego pliku źródłowego jako modułu:

Składnia instrukcji importu

```
import modname [as varname] [, ...]
```

- Powyżej:

modname | może być nazwą modułu lub sekwencją identyfikatorów separowanych kropkami, odpowiadającą nazwie modułu w pakiecie

varname | jest opcjonalnym identyfikatorem obiektu modułu

Zawartość modułu

- **Ciałem modułu** jest sekwencja instrukcji zawartych w pliku modułu
- Nie ma żadnej specjalnej składni wskazującej, że dany plik jest modułem – może nim być dowolny, poprawny plik Pythona
- Ciało modułu jest wykonywane podczas pierwszego importu
- Wtedy tworzony jest obiekt modułu i wpisywany do słownika *sys.modules*
- Globalna przestrzeń nazw modułu jest sukcesywnie zapełniana podczas wykonywania modułu

Atrybuty obiektów modułów

- Instrukcja importu tworzy nową przestrzeń nazw zawierającą wszystkie atrybuty modułu
- Atrybuty są dostępne poprzez nazwę modułu lub jej alias (nazwę obiektu modułu):

Dostęp do atrybutów poprzez nazwę modułu

```
>>> import mymodule  
>>> a = mymodule.f()
```

Dostęp do atrybutów poprzez alias

```
>>> import mymodule as alias  
>>> a = alias.f()
```

Atrybuty obiektów modułów

- Celem ciała modułu jest **utworzenie atrybutów obiektu modułu**
 - instrukcje `def` tworzą i wiążą obiekty funkcji
 - instrukcje `class` tworzą i wiążą obiekty klas
 - instrukcje przypisania wiążą atrybuty dowolnego typu
- Dobrą praktyką jest nie wykonywanie żadnych innych funkcjonalności na poziomie modułu

Standardowe atrybuty modułu

- Instrukcja importu ustawia kilka standardowych atrybutów modułu podczas jego tworzenia (jeszcze przed wykonaniem ciała modułu), m.in.:

ATRYBUT	ZNACZENIE
<u>__dict__</u>	obiekt słownika wykorzystywany jako przestrzeń nazw modułu atrribut nie jest dostępny z poziomu kodu wszystkie wpisy w słowniku są dostępne dla kodu jako atrybuty globalne modułu
<u>__name__</u>	nazwa modułu
<u>__file__</u>	nazwa pliku z którego moduł został załadowany
<u>__doc__</u>	dokumentacja modułu

- Jeżeli pierwsza instrukcja w module będzie literałem tekstowym, to Python zwiąże ten tekst z atrybutem __doc__ modułu

Obiekty wbudowane Pythona

- Python udostępnia wiele **obiektów wbudowanych** (*built-in objects*)
- Wszystkie te obiekty są atrybutami modułu *builtins*
- Gdy Python ładuje moduł, tworzony jest atrybut o nazwie `__builtins__`, który odnosi się do modułu *builtins* lub jego katalogu
- Podczas odwołania do zmiennej, jest ona poszukiwana kolejno:
 - w lokalnej przestrzeni nazw
 - w globalnej przestrzeni nazw modułu
 - w `__builtins__` bieżącego modułu
 - zgłaszanego jest wyjątek *NameError*

Obiekty wbudowane Pythona – przykład

Przedefiniowanie funkcji wbudowanej

```
>>> # abs przyjmuje argument numeryczny
... # teraz przyjmie także argument tekstowy
... import builtins

>>> _abs = builtins.abs # zapamiętujemy oryginalną funkcję

>>> def abs(str_or_num):
...     if isinstance(str_or_num, str): # gdy argument jest
...                                     # tekstowy...
...         return ''.join(sorted(set(str_or_num)))
...     return _abs(str_or_num) # wywołanie oryginalne
...
>>> builtins.abs = abs # zamiana oryginalnej funkcji na nową
```

Instrukcja *from*

- Instrukcja **from** umożliwia zaimportowanie wybranych atrybutów do przestrzeni nazw

Składnia instrukcji *from*

```
from modname import attrname [as varname] [, ...]
```

- Parametr *modname* może być sekwencją identyfikatorów separowanych kropkami – umożliwia to wskazanie modułu w ramach pakietu

Instrukcja *from*

- Instrukcja **from** może także wykorzystać następującą składnię:

Składnia instrukcji *from*

```
from modname import *
```

- Standardowo polecenie wiąże wszystkie atrybuty modułu, poza tymi, których nazwy rozpoczynają się od znaku podkreślenia _ (atrybuty ukryte/prywatne)
- Jeśli moduł zawiera atrybut o nazwie all, to importowane są tylko te atrybuty, których nazwy znajdują się na liście tego atrybutu

Program główny

- Uruchomienie programu w Pythonie rozpoczyna się od wykonania **skryptu najwyższego poziomu** (*top-level script*) zw. programem głównym
- Jest on wykonywany, jak każdy inny moduł, z tą różnicą, że kod bajtowy jest trzymany w pamięci i nie jest zapisywany na dysk
- Moduł programu głównego ma zawsze nazwę main

Program główny

- Jeśli jakiś kod ma być wykonany, tylko wtedy, gdy jest uruchamiany z programu głównego, to należy go umieścić w bloku instrukcji warunkowej:

Kod uruchamiany z programu głównego

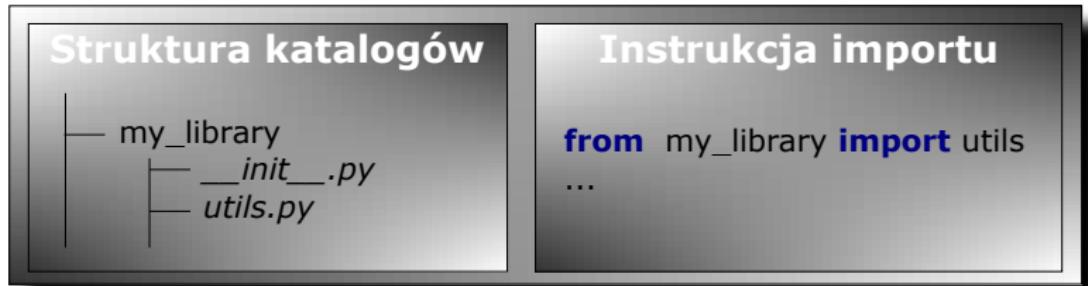
```
if __name__ == '__main__':
    # kod wykonywany tylko z poziomu programu głównego
```

Pakiety

- **Pakiet** (*package*) – jest modułem zawierającym inne moduły
- Pakiet może zawierać **podpakiety** (*subpackages*), tworząc hierarchiczną strukturę drzewa
- Nie ma ograniczenia na głębokość zagnieżdżenia
- Pakiet znajduje się w podkatalogu o tej samej nazwie, katalogu wymienionym w *sys.path* lub w pliku ZIP

Pakiety

- Ciało modułu pakietu znajduje się w pliku `__init__.py`
- Plik jest obowiązkowy, nawet jeśli jest pusty – powoduje, że katalog jest traktowany jak pakiet Pythona, a nie zwykły katalog



- Podpakiety muszą posiadać własne pliki `__init__.py`

Pakiety

- Moduł M z pakietu P można zimportować jako $P.M$
- Więcej kropek umożliwia nawigację po hierarchicznej strukturze pakietów
- Plik `__init__.py` jest ładowany zanim zostanie załadowany jakikolwiek moduł tego pakietu

Atrybuty specjalne obiektu pakietu

ATRYBUT	ZNACZENIE
<code>__file__</code>	ścieżka prowadząca do ciała modułu pakietu, tzn. pliku <code>__init__.py</code>
<code>__package__</code>	nazwa pakietu
<code>__path__</code>	lista ścieżek do katalogów z których są ładowane moduły i podpakiety

- W pliku `__init__.py` można ustawić globalną zmienną `__all__`, aby kontrolować, co się stanie po wykonaniu polecenia
*from <package-name> import **

Ćwiczenia/przykłady

- Ćwiczenie/przykład 6.1:
Użycie modułów i pakietów



ĆWICZENIA

Plan szkolenia

- 1 WPROWADZENIE DO JĘZYKA PYTHON
- 2 PODSTAWOWE KONCEPCJE
- 3 ZŁOŻONE TYPY DANYCH
- 4 PROGRAMOWANIE FUNKCYJNE
- 5 KLASY I OBIEKTY
- 6 MODUŁY I PAKIETY
- 7 OPERACJE NA PLIKACH**
- 8 WYJĄTKI
- 9 WAŻNE WBUDOWANE MODUŁY I BIBLIOTEKI

7 OPERACJE NA PLIKACH

- menedżer kontekstu
- pakiet `io`
- otwieranie plików
- zamykanie plików
- atrybuty i metody strumieni



Zarządzanie zasobami

- Nieprawidłowe zarządzanie zasobami (plikami, połączeniami sieciowymi, połączeniami do bazy danych, itp.) może prowadzić do **wycieków**
- Zasoby z reguły wymagają wykonania pewnych operacji przed i po ich użyciu (zamknięcia, zwrócenia obiektu do puli, itp.)
- Wypełnienie tych wymogów jest czasami kłopotliwe i wymusza przykładanie się do staranności w tworzeniu kodu
- Konstrukcja **menedżera kontekstu** może znaczaco uprościć ten proces

Menedżer kontekstu

- Klasa **menedżera kontekstu** (*context manager*) musi posiadać zdefiniowane dwie metody specjalne:

METODA	DZIAŁANIE
<code>__enter__(self)</code>	wywoływana przed użyciem zasobu
<code>__exit__(self, etype, evalue, etraceback)</code>	wywoływana po zakończeniu pracy z zasobem jeśli ciało nie zgłosi wyjątku, to metoda jest wywoływana z 3 argumentami <i>None</i>

- Użycie menedżera kontekstu upraszcza instrukcję **with**

Składnia instrukcji *with*

```
with expression [as varname]:  
    statement(s)
```

- Menedżer kontekstu jest zwracany przez wyrażenie *expression*

Definicja menedżera kontekstu – przykład

Definicja menedżera kontekstu

```
>>> class Tag:  
...     def __init__(self, tag_name):  
...         self.tag_name = tag_name  
...     def __enter__(self):  
...         print(f'<{self.tag_name}>')  
...     def __exit__(self, etype, evalue, etraceback):  
...         print(f'</ {self.tag_name}>')  
...     ...
```

Użycie menedżera kontekstu – przykład

Użycie menedżera kontekstu

```
>>> html_tag = Tag('HTML')
>>> body_tag = Tag('BODY')
>>> with html_tag, body_tag:
...     print('treść dokumentu')
...
<HTML>
<BODY>
treść dokumentu
</BODY>
</HTML>
```

Pakiet *io*

- Odczyt i zapis danych do plików to jedne z najbardziej podstawowych operacji w programowaniu
- Można pracować z plikami tekstowymi i binarnymi
- Pakiet *io* – standardowy pakiet umożliwiający realizację podstawowych **operacji wejścia-wyjścia** (*IO – Input/Output*)
- Operacje odczytu i zapisu są wykonywane na **obiektach plików** (*file objects* lub *file-like objects*)
- Obiekty plików są inaczej nazywane **strumieniami** (*streams*)

Rodzaje operacji we/wy

- Istnieją 3 główne kategorie operacji we/wy:
 - tekstowe (*text I/O*)
 - oczekują i tworzą obiekty typu *str*
 - dane binarne podlegają transparentnie kodowaniu/dekodowaniu
 - podobnie znaki nowych linii specyficzne dla platformy
 - binarne (*binary I/O, buffered I/O*)
 - oczekują obiektów bajtowych i tworzą obiekty typu *bytes*
 - żadne kodowanie/dekodowanie ani translacja znaków nie jest dokonywana
 - niskopoziomowe (*raw I/O, unbuffered I/O*)
 - rzadko wykorzystywane z poziomu kodu programistów

Otwieranie obiektów plików

- Funkcja `io.open` jest synonimem wbudowanej funkcji `open`

Otwieranie strumienia

```
open(file, mode='r', buffering=-1, encoding=None, errors=None,  
newline=None, closefd=True, opener=None)
```

- Funkcja zwraca obiekt pliku
- Parametr `file` – ścieżka do pliku lub deskryptor pliku (wartość całkowita zwrócona przez funkcję `os.open`)
- Funkcja może nie tylko otworzyć plik, ale także go utworzyć
- W przypadku podania deskryptora, plik musi być już otwarty

Tryby otwarcia pliku

PARAMETR mode	DZIAŁANIE
'r'	<ul style="list-style-type: none">* plik jest otwierany do odczytu (wartość domyślna)* strumień jest ustawiany na początek pliku
'r+'	<ul style="list-style-type: none">* jak 'r', ale plik jest otwierany do aktualizacji (odczytu i zapisu)
'w'	<ul style="list-style-type: none">* plik jest otwierany do zapisu* strumień jest ustawiany na początek pliku* jeśli plik istnieje, to jego zawartość jest obcinana, w przeciwnym razie plik jest tworzony
'w+'	<ul style="list-style-type: none">* jak 'w', ale plik jest otwierany do aktualizacji (odczytu i zapisu)
'x'	<ul style="list-style-type: none">* jak 'w', ale w trybie wyłączności tworzenia pliku (nowy plik zostanie utworzony, jeśli nie istnieje, w przeciwnym razie wystąpi wyjątek)
'x+'	<ul style="list-style-type: none">* jak 'w', ale plik jest otwierany do aktualizacji (odczytu i zapisu) w trybie wyłączności tworzenia pliku
'a'	<ul style="list-style-type: none">* plik jest otwierany do zapisu* jeśli plik istnieje, to nowa treść jest dopisywana na końcu, w przeciwnym razie plik jest tworzony
'a+'	<ul style="list-style-type: none">* jak 'a', ale plik jest otwierany do aktualizacji (odczytu i zapisu)
Każdy z powyższych trybów można jeszcze uzupełnić o...	
'b'	<ul style="list-style-type: none">* tryb binarny
't'	<ul style="list-style-type: none">* tryb tekstowy (wartość domyślna)

Buforowanie

- Parametr *buffering* określa **politykę buforowania**

PARAMETR buffering	DZIAŁANIE
0	oznacza brak buforowania (dozwolone tylko w trybie binarnym)
1	oznacza buforowanie linii (użyteczne w trybie tekstowym)
> 1	oznacza stały bufor o podanej wielkości
< 0 lub brak	oznacza przyjęcie wartości domyślnej (dla konsoli interaktywnych – wielkość linii, dla pozostałych – wartość <code>io.DEFAULT_BUFFER_SIZE</code>)

Zamykanie plików

- Każdy otwarty plik, po zakończeniu pracy z nim, należy zamknąć
- Otwarcie pliku powoduje utworzenie przez system deskryptora pliku
- Systemy operacyjne mają ograniczenia na liczbę otwartych plików w danym procesie

Zbyt duża liczba otwartych plików

```
>>> pliki = []
>>> for _ in range(100_000):
...     pliki.append(open('test.txt', 'w'))
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
OSError: [Errno 24] Too many open files: 'test.txt'
```

Zamykanie plików

- Obiekty plików, zwracane przez funkcję `open` są menedżerami kontekstu
- Stąd użycie instrukcji `with` może zagwarantować automatyczne zamknięcie pliku, bezpośrednio po zakończeniu wykonywania ciała instrukcji

Atrybuty i metody obiektów plików

METODA/ATRYBUT	DZIAŁANIE
<code>f.close()</code>	zamyka plik
<code>f.closed</code>	atrybut tylko-do-odczytu weryfikujący, czy metoda <code>close</code> została wywołana
<code>f.encoding</code>	atrybut tylko-do-odczytu z nazwą kodowania (nie istnieje dla strumieni binarnych)
<code>f.flush()</code>	żądanie zapisu zawartości bufora do systemu operacyjnego
<code>f.isatty()</code>	sprawdza, czy plik jest interaktywnym terminaliem
<code>f.fileno()</code>	zwraca deskryptor pliku
<code>f.mode</code>	atrybut tylko-do-odczytu zawierający tryb otwarcia pliku
<code>f.name</code>	atrybut tylko-do-odczytu zawierający nazwę pliku lub wartość deskryptora
<code>f.read(size=-1)</code>	czyta maksymalnie <code>size</code> bajtów lub znaków (zależnie od typu strumienia) gdy <code>size < 0</code> , czytana jest cała zawartość pliku
<code>f.readline(size=-1)</code>	czyta jedną linię tekstu lub maksymalnie zadaną liczbę bajtów
<code>f.readlines(size=-1)</code>	czyta zawartość całego pliku i zwraca ją w postaci listy lub zwraca maksymalnie zadaną liczbę bajtów

Atrybuty i metody obiektów plików

METODA	DZIAŁANIE
<code>f.seek(pos, how=io.SEEK_SET)</code>	zmienia bieżącą pozycję w pliku na wartość odległą o <i>pos</i> w stosunku do punktu odniesienia <i>how</i> punkty odniesienia: SEEK_SET (początek), SEEK_CUR (pozycja bieżąca), SEEK_END (koniec)
<code>f.tell()</code>	zwraca bieżącą pozycję
<code>f.truncate([size])</code>	obcinà plik, by zawierał maksymalnie <i>size</i> bajtów w razie braku argumentu, jego rolę pełni wartość zwrócona przez <i>tell</i>
<code>f.write(s)</code>	wpisuje podany tekst do pliku
<code>f.writelines(lst)</code>	równoważne poleceniu: <i>for line in lst: f.write(line)</i>

Ćwiczenia/przykłady

- Ćwiczenie/przykład 7.1:
Metamorfoza – odczyt zawartości plików
- Ćwiczenie/przykład 7.2:
Łączenie plików
- Ćwiczenie/przykład 7.3:
Utrwalanie i odtwarzanie obiektów



Plan szkolenia

- 1 WPROWADZENIE DO JĘZYKA PYTHON
- 2 PODSTAWOWE KONCEPCJE
- 3 ZŁOŻONE TYPY DANYCH
- 4 PROGRAMOWANIE FUNKCYJNE
- 5 KLASY I OBIEKTY
- 6 MODUŁY I PAKIETY
- 7 OPERACJE NA PLIKACH
- 8 WYJĄTKI
- 9 WAŻNE WBUDOWANE MODUŁY I BIBLIOTEKI

8 WYJĄTKI

- wyjątki – teoria
- obsługa wyjątków
- standardowe klasy wyjątków
- własne klasy wyjątków
- asercje



Wyjątki

- **Wyjątki** (*exceptions*) – obiekty wykorzystywane do sygnalizacji błędów i sytuacji wyjątkowych (anomalii w działaniu)
- Sygnalizacja błędu polega na **wyrzuceniu** (*raise*) wyjątku i jego dalszej propagacji
- **Obsługa** (*handling*) wyjątku oznacza przechwycenie obiektu wyjątku z mechanizmu propagacji i wykonanie akcji “naprawy” – usunięciu skutków powstałego problemu

Wyjątki

- Jeżeli program nie obsłuży wyjątku, to zakończy się z komunikatem opisu **śladu stosu** (*traceback*)
- Jeśli natomiast wyjątek zostanie obsłużony, to program będzie kontynuował działanie, tak, jakby żadnego błędu nie było

Instrukcja `try`

- Mechanizmu obsługi wyjątków dostarcza instrukcja `try`

Składnia instrukcji `try`

```
try:  
    statement(s)  
except [expression [as target]]:  
    statement(s)  
[else:  
    statement(s)]
```

- Klauzula `try` zawiera blok instrukcji, w trakcie wykonania których może wystąpić wyjątek
- Jest nazywana **blokiem chronionym** (*guarded clause*)

Klauzula `except`

- Blok instrukcji zawarty w klauzuli `except` jest nazywany **obsługą wyjątku** (*exception handler*)
- Kod jest wykonywany, gdy wyrażenie zawarte w klauzuli `except` odpowiada typowi wyjątka propagowanego z klauzuli `try`
- Wyrażenie *expression* jest klasą lub krotką klas
- Obiekt wyjątku powinien należeć do jednej z tych klas lub ich podklas

Klauzula `except`

- Opcjonalny identyfikator `target` jest nazwą zmiennej z którą Python wiąże obiekt wyjątku (zanim zostanie wykonany blok handlera)
- Handler może uzyskać dostęp do obiektu wyjątku także poprzez wywołanie funkcji `exc_info` z modułu `sys`

Przykład

Przykład instrukcji *try*

```
>>> try:  
...     1/0  
... except ZeroDivisionError:  
...     print('wykryta próba dzielenia przez zero')  
...  
wykryta próba dzielenia przez zero  
>>>
```

Handlery

- Jeżeli instrukcja `try` zawiera wiele klauzul `except`, mechanizm propagacji wyjątków testuje klauzule w kolejności ich podania
- W celu obsługi wyjątku wykonywany jest blok instrukcji (handler) pierwszej klauzuli `except`, której wyrażenie pasuje do obiektu wyjątku
- Z tego powodu handlery dla bardziej szczegółowych wyjątków powinny poprzedzać te dla bardziej ogólnych

- Ostatnia klauzula **except** może pominać wyrażenie – przechwyci ona **każdy wyjątek, który wcześniej nie został obsłużony**
- Taka bezwarunkowa obsługa jest rzadko stosowana – w ogólności takiej praktyki należy się wystrzegać
- Można jej użyć w przypadku funkcji-wrapperów, które muszą wykonać dodatkowe operacje przed ponownym wyrzuceniem wyjątku

Propagacja wyjątku

- Propagacja wyjątku kończy się w momencie odnalezienia handlера, którego wyrażenie pasuje do obiektu propagowanego wyjątku

Propagacja wyjątku

```
>>> try:  
...     try:  
...         1/0  
...     except:  
...         print('przechwycono wyjątek')  
... except ZeroDivisionError:  
...     print('wykryta próba dzielenia przez zero')  
...  
przechwycono wyjątek
```

Klauzula `else`

- Opcjonalna klauzula `else` wykonywana jest tylko wtedy, gdy klauzula `try` zakończyła się w normalny sposób
- Klauzula nie wykona się, jeśli:
 - wyjątek był propagowany z klauzuli `try`
 - z klauzuli `try` nastąpiło wyjście za pomocą instrukcji: `break`, `continue` lub `return`

Klauzula *else*

Klauzula *else*

```
>>> def test(value):
...     print(repr(value), 'to', end=' ')
...     try:
...         value + 0
...     except TypeError:
...         try:
...             value + ''
...         except TypeError:
...             print('ani liczba, ani tekst')
...         else:
...             print('jakiś tekst')
...     else:
...         print('jakaś liczba')
...
...
```

Klauzula *else*

Klauzula *else*

```
>>> test(True)  
True to jakąś liczba
```

```
>>> test('abc')  
'abc' to jakiś tekst
```

```
>>> test((1, 2))  
(1, 2) to ani liczba, ani tekst
```

Instrukcja *try*

- Instrukcja **try** posiada jeszcze inną formę składniową:

Składnia instrukcji *try*

```
try:  
    statement(s)  
finally:  
    statement(s)
```

- W tym wariantie instrukcja nie posiada klauzuli **except**, ale posiada dokładnie jedną klauzulę **finally**

Klauzula ***finally***

- Kod klauzuli ***finally*** wykonuje się zawsze, po kodzie klauzuli ***try***, niezależnie od sposobu w jaki się ona zakończyła
- Stąd typowym zastosowaniem dla tej konstrukcji jest zagwarantowanie wykonania **kodu finalizacyjnego** który musi być wykonany, niezależnie od tego, czy wyjątek wystąpi, czy też nie (*clean-up handler*)
- Kod klauzuli ***finally*** wykona się przed ewentualną dalszą propagacją nieobsłużonego wyjątku

Klauzula ***finally***

- W klauzuli ***finally*** nie może się znaleźć instrukcja ***continue***
- Wystąpienie instrukcji ***break*** i ***return*** jest dopuszczalne, ale nie zalecane – ich użycie spowoduje przerwanie propagacji wyjątku

Instrukcja *try*

- Instrukcja *try-finally* może także zawierać klauzule **except**:

Składnia instrukcji *try*

```
try:  
    # blok chroniony  
    ...  
except expression:  
    # kod handlера  
    ...  
finally:  
    # kod "sprzątający"  
    ...
```

Instrukcja `try`

- Wtedy jest ona równoważna konstrukcji:

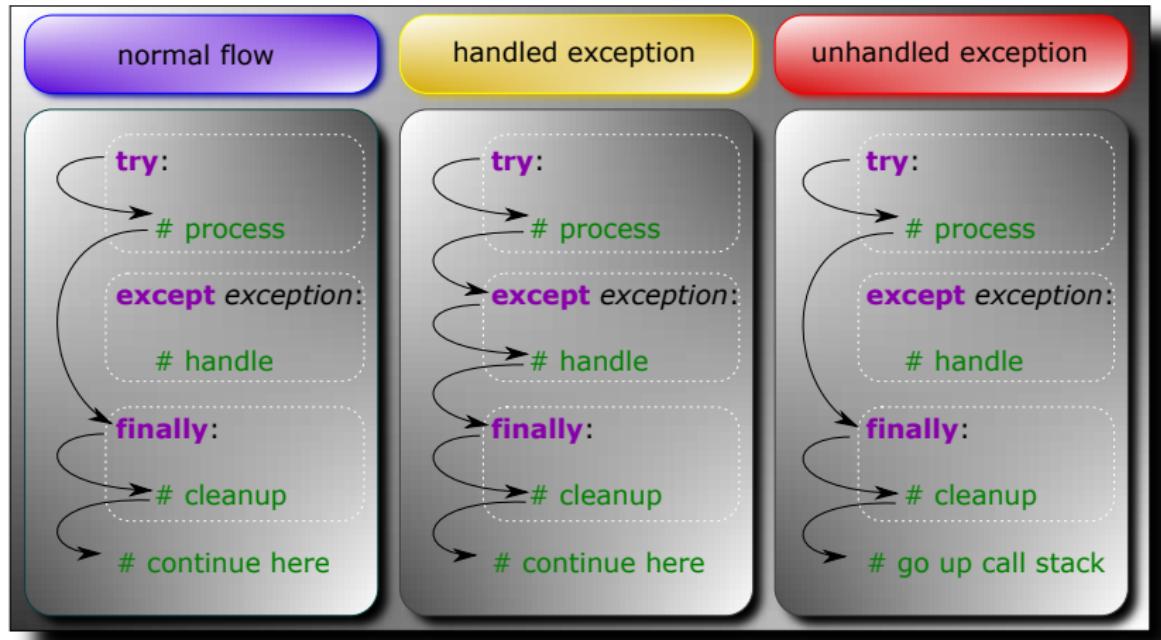
Składnia instrukcji `try`

```
try:  
    try:  
        # blok chroniony  
        ...  
    except expression:  
        # kod handlera  
        ...  
finally:  
    # kod "sprzątający"  
    ...
```

- Instrukcja `try` może zawierać wiele klauzul `except` i opcjonalnie jedną klauzulę `else`

Przepływ sterowania

- Wyjątki mogą zmienić przepływ sterowania w programie



Wyrzucanie wyjątków

- Do samodzielnego wyrzucenia wyjątku służy instrukcja **raise**

Wyrzucenie wyjątku

```
raise exception(args)
```

- Jest ona stosowana w celu **wyrzucenia wyjątku podanego typu**
- Może to być obiekt wyjątku wbudowanego lub własnego
- Parametr *args* jest krotką argumentów wykorzystywanych do utworzenia obiektu wyjątku

Łańcuchy wyjątków

Wyrzucenie wyjątku

```
raise exception(args) from original_exception
```

- To polecenie umożliwia **łączenie wyjątków w łańcuchy** (*exception chaining*)
- Parametr *original_exception* odnosi się do oryginalnego obiektu wyjątku – faktycznej przyczyny
- Oryginalny wyjątek zostanie przypisany do atrybutu *__cause__* wyjątku wyrzuconego

Łańcuchy wyjątków

Tworzenie łańcucha wyjątków

```
>>> class InvalidDataError(Exception): pass  
...  
>>> def process(data):  
...     try:  
...         i = int(data)  
...     except ValueError as err:  
...         raise InvalidDataError('invalid data') from err  
...
```

Wyrzucanie wyjątków

Wyrzucenie wyjątku

`raise`

- Pusta instrukcja `raise` może wystąpić tylko w obsłudze wyjątku (handlerze) lub funkcji z tego miejsca wywołanej
- Jej wykonanie spowoduje **ponowne wyrzucenie przechwyconego wyjątku**
- Takie postępowanie jest uzasadnione, jeśli handler nie jest w stanie sensownie obsłużyć wyjątku lub obsługuje go tylko częściowo – wtedy należy go dalej propagować

Standardowe klasy wyjątków

KLASA WYJĄTKU	OPIS
<i>AssertionError</i>	asercja zawiodła
<i>AttributeError</i>	błędna referencja do atrybutu lub błędne przypisanie wartości
<i>FloatingPointError</i>	błąd operacji zmiennoprzecinkowej (rozszerza klasę <i>ArithmeticError</i>)
<i>IOError</i>	<i>IOError</i> błąd operacji we/wy (synonim <i>OSError</i>)
<i>ImportError</i>	błąd instrukcji importu (nieodnaleziony moduł lub importowana nazwa)
<i>IndentationError</i>	parser natrafił na błąd składniowy nieprawidłowego formowania w bloki - błędne wcięcia (rozszerza klasę <i>SyntaxError</i>)
<i>IndexError</i>	błędny indeks w sekwencji, wykraczający poza zakres (rozszerza klasę <i>LookupError</i>)
<i>KeyError</i>	użycie klucza, który nie istnieje w słowniku (rozszerza klasę <i>LookupError</i>)
<i>KeyboardInterrupt</i>	została wciśnięta kombinacja klawiszy przerywająca operację (<Ctrl/C>, <Ctrl/Break>, <Delete>, itp. zależnie od platformy)
<i>MemoryError</i>	operacja wyczerpała dostępną pamięć
<i>NameError</i>	odwołanie do zmiennej niezwiązanej z wartością

Standardowe klasy wyjątków

KLASA WYJĄTKU	OPIS
<i>NotImplementedError</i>	wyjątek zgłoszany przez bazową klasę abstrakcyjną do zasygnalizowania, że podklasa musi nadpisać metodę
<i>OSError</i>	wyjątek zgłoszany przez funkcje modułu <i>os</i> do zasygnalizowania błędów zależnych od platformy
<i>OverflowError</i>	wynik operacji arytmetycznej wykracza poza wymagany zakres (rozszerza klasę <i>ArithmeticError</i>)
<i>SyntaxError</i>	parser natrafił na błąd składniowy
<i>SystemError</i>	błąd wewnętrzny Pythona lub modułu rozszerzającego
<i>TypeError</i>	operacja lub funkcja została wykonana na obiekcie niewłaściwego typu
<i>UnboundLocalError</i>	nastąpiło odwołanie do lokalnej zmiennej, która aktualnie nie jest powiązana z wartością (rozszerza klasę <i>NameError</i>)
<i>ValueError</i>	operacja lub funkcja została wywołana na obiekcie właściwego typu, ale z nieprawidłowymi wartościami
<i>ZeroDivisionError</i>	zerowa wartość dzielnika operacji dzielenia i modulo lub drugiego argumentu funkcji <i>divmod</i> (rozszerza klasę <i>ArithmeticError</i>)

Ćwiczenia/przykłady

- Ćwiczenie/przykład 8.1:
Osoby – użycie wyjątków standardowych



Hierarchia wyjątków

- Klasą bazową dla wszystkich wyjątków w Pythonie jest klasa *BaseException*
- Ich źródłem może być interpreter, wbudowane funkcje lub kod użytkownika
- Większość klas wyjątków rozszerza typ *Exception*
- Oprócz klasy *Exception*, po klasie *BaseException* dziedziczą jeszcze:

KLASA WYJĄTKU	OPIS
<i>KeyboardInterrupt</i>	wyjątki powodowane przez sekwencje klawiszy przerywających akcje, np. <Ctrl/C>, <Ctrl/Break>
<i>GeneratorExit</i>	wyjątki wywoływane podczas zamykania generatora
<i>SystemExit</i>	wyjątki zwykle powodowane przez wywołanie funkcji exit z modułu sys

Tworzenie własnych klas wyjątków

- **Własne klasy wyjątków** należy tworzyć poprzez rozszerzenie klasy *Exception* lub dowolnej z jej podklas
- Często ogranicza się to tylko do zdefiniowania *docstring'a*

Własna klasa wyjątku

```
>>> class MyException(Exception):
...     def __init__(self, msg):
...         self.msg = msg
...     def __str__(self):
...         return repr(self.msg)
...
>>> class SpecificException(MyException):
...     """
...     Custom specific exception class
...     """
...
... 
```

Błędy w programie

- Wyjątki mogą sygnalizować błędy wykonania
- Ich brak nie oznacza, że program działa prawidłowo
- Mogą w nim występować błędy semantyczne związane poprawną implementacją błędного algorytmu (program liczy, ale nie to, co powinien)
- W celu uniknięcia tego typu problemów należy:
 - uzupełnić kod o testy
 - użyć mechanizmu asercji do zdefiniowania warunków wstępnych i końcowych

Asercje

- **Asercje (assertions)** służą do weryfikacji **oczekiwanej stanu wewnętrznego aplikacji**
- Jeżeli w aplikacji możemy wskazać miejsca w których w danym momencie warunek ma ustaloną wartość, to znaczy, że w aplikacji istnieją **niezmienniki (invariants)**
- Asercje są wygodnym narzędziem do sprawdzenia niezmienników
- Instrukcja asercji ma postać:

Składnia asercji

```
assert condition [, expression]
```

Asercje

- Jeżeli podany predykat *condition* będzie miał wartość **False**, to spowoduje to wyrzucenie wyjątku **AssertionError**
- W takiej sytuacji opcjonalne wyrażenie *expression* zostanie przekazane jako argument do klasy wyjątku i stanie się komunikatem błędu

Przykład asercji

```
>>> assert False, 'nieoczekiwana sytuacja'  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
AssertionError: nieoczekiwana sytuacja
```

Zmienna __debug__

- Aby uruchomić aplikację **bez asercji** można:
 - użyć opcji `-O` (optymalizacja wygenerowanego kodu bajtowego)
 - ustawić zmienną środowiska `PYTHONOPTIMIZE=O`
- Spowoduje to usunięcie asercji z kodu bajtowego
- Uruchomienie aplikacji z opcją `-O` spowoduje także ustawienie zmiennej wbudowanej __debug__ na wartość **False**
(w przeciwnym razie zmienna ma wartość **True**)
- Można to wykorzystać m.in. do usunięcia z kodu bajtowego definicji funkcji z których korzystają asercje

Zmienna `__debug__`

Skrypt `program.py`

```
n = 13
if __debug__:
    def isLucky(number): return number != 13
assert isLucky(n), 'wylosowałeś 13'
print('Twoja szczęśliwa liczba to', n)
```

Uruchomienie skryptu

```
> python program.py
Traceback (most recent call last):
  File "program.py", line 5, in <module>
    assert isLucky(n), 'wylosowałeś 13'
AssertionError: wylosowałeś 13

> python -O program.py
Twoja szczęśliwa liczba to 13
```

Ćwiczenia/przykłady



- Ćwiczenie/przykład 8.2:
Osoby – własne klasy wyjątków
- Ćwiczenie/przykład 8.3:
Telefon na kartę – sytuacje wyjątkowe

ĆWICZENIA

Plan szkolenia

- 1 WPROWADZENIE DO JĘZYKA PYTHON
- 2 PODSTAWOWE KONCEPCJE
- 3 ZŁOŻONE TYPY DANYCH
- 4 PROGRAMOWANIE FUNKCYJNE
- 5 KLASY I OBIEKTY
- 6 MODUŁY I PAKIETY
- 7 OPERACJE NA PLIKACH
- 8 WYJĄTKI
- 9 WAŻNE WBUDOWANE MODUŁY I BIBLIOTEKI



9 WAŻNE WBUDOWANE MODUŁY I BIBLIOTEKI

- typy wbudowane
- funkcje wbudowane
- biblioteka standardowa
- moduły wbudowane
- popularne biblioteki

TEORIA

Elementy wbudowane

- Termin “**wbudowany**” (*built-in*) ma wiele znaczeń w Pythonie
- W większości kontekstów oznacza: *obiekt dostępny bezpośrednio dla kodu, bez konieczności jego importowania*

Typy wbudowane

- **Wbudowane typy danych** zostały omówione w poprzednich rozdziałach:
 - typy numeryczne
 - sekwencje (łańcuchy tekstowe, listy, krotki, zakresy)
 - zbiory
 - słowniki
 - klasy i instancje
 - wyjątki
- Python zawiera także wiele **wbudowanych funkcji**

Funkcje wbudowane

FUNKCJA WBUDOWANA	OPIS
bool([x])	zwraca wartość typu logicznego
int(x=0) int(x, base=10)	zwraca wartość liczbową całkowitą
float([x])	zwraca liczbę zmienoprzecinkową na podstawie podanej wartości numerycznej lub tekstowej
complex([real[, imag]])	zwraca liczbę zespoloną
str(object="") str(object=b", encoding='utf-8', errors='strict')	zwracainstancję typu tekstowego
bytearray([source[, encoding[, errors]]])	zwraca nową tablicę bajtów
bytes([source[, encoding[, errors]]])	zwraca nową instancję typu <i>bytes</i>
tuple([iterable])	zwraca krotkę
slice(stop) slice(start, stop[, step])	zwraca instancję wycinka reprezentującą zbiór elementów określonych przez zakres
list([iterable])	zwraca obiekt listy
set([iterable])	zwraca instancję zbioru
frozenset([iterable])	zwraca instancję zbioru typu <i>frozenset</i>
dict(**kwarg) dict(mapping, **kwarg) dict(iterable, **kwarg)	zwraca nowy słownik

Funkcje wbudowane

FUNKCJA WBUDOWANA	OPIS
object()	zwraca nową instancję typu <i>object</i>
type(object) type(name, bases, dict)	w wersji jednoargumentowej zwraca typ obiektu w wersji trójargumentowej jest dynamicznym odpowiednikiem deklaracji nowego typu
memoryview(object)	tworzy instancję typu <i>memoryview</i> odnoszącą się do podanego obiektu obiekt musi implementować protokół buforowania umożliwia dostęp do stanu wewnętrznego obiektu, bez konieczności kopiowania
property(fget=None, fset=None, fdel=None, doc=None)	tworzy właściwość
bin(x)	konwertuje liczbę całkowitą na tekst reprezentujący wartość binarną (z przedrostkiem 0b)
oct(x)	konwertuje liczbę całkowitą na tekst reprezentujący wartość w układzie ósemkowym (z przedrostkiem 0o)
hex(x)	konwertuje liczbę całkowitą na tekst reprezentujący wartość heksadecymalną (z przedrostkiem 0x)
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)	wpisuje obiekty do plikowego strumienia tekstuowego, oddzielając je separatorem <i>sep</i> i kończąc łańcuchem <i>end</i>
input([prompt])	wyświetla podaną podpowiedź, wczytuje dane z wejścia i konwertuje na tekst

Funkcje wbudowane

FUNKCJA WBUDOWANA	OPIS
repr (object)	zwraca reprezentację tekstową obiektu, na podstawie której można odtworzyć obiekt za pomocą funkcji <i>eval</i>
ascii (object)	podobnie jak <i>repr</i> , zwraca reprezentację tekstową obiektu, ale maskuje znaki nie-ASCII za pomocą \x, \u lub \U
eval (expr, globals=None, locals=None)	umożliwia przeparsowanie i ewaluację podanego wyrażenia (zwracającą wartością jest <i>None</i>)
exec (statement, [globals[, locals]])	umożliwia dynamiczne wykonanie bloku kodu (zwracaną wartością jest <i>None</i>)
compile (source, filename, mode, flags=0, dont_inherit=False, optimize=-1)	kompiluje kod, który można wykonać za pomocą funkcji <i>exec</i> lub <i>eval</i>
getattr (object, name[, default])	zwraca wartość nazwanego atrybutu podanego obiektu
setattr (object, name, value)	tworzy nowy lub aktualizuje istniejący atrybut
hasattr (object, name)	testuje, czy obiekt posiada atrybut o podanej nazwie
delattr (object, name)	usuwa atrybut o podanej nazwie
breakpoint (*args, **kwds)	uruchamia debugger w miejscu wywołania funkcji (od Pythona 3.7)

Funkcje wbudowane

FUNKCJA WBUDOWANA	OPIS
abs(x)	zwraca wartość bezwzględną
divmod(a, b)	zwraca krotkę ($a // b$, $a \% b$)
sum(iterable[, start])	zwraca sumę elementów <i>iterable</i>
pow(x, y[, z])	zwraca x^y lub $x^y \% z$
min(iterable, *[, key, default]) min(arg1, arg2, *args[, key])	zwraca najmniejszy element
max(iterable, *[, key, default]) max(arg1, arg2, *args[, key])	zwraca największy element
all(iterable)	zwraca wartość <i>True</i> , jeśli wszystkie elementy <i>iterable</i> mają wartość logiczną <i>True</i> lub <i>False</i> – gdy sekwencja <i>iterable</i> jest pusta
any(iterable)	zwraca wartość <i>True</i> , jeśli jakikolwiek element <i>iterable</i> ma wartość logiczną <i>True</i> (gdy <i>iterable</i> jest pusta – zwracana jest wartość <i>False</i>)
ord(c)	zwraca kod znaku jego reprezentacji tekstowej
chr(code)	zwraca tekst zawierający znak o podanym kodzie Unicode
round(number[, ndigits])	zwraca zaokrągloną liczbę z precyzją <i>ndigits</i> cyfr
format(value[, format_spec])	zwraca sformatowaną reprezentację podanej wartości

Funkcje wbudowane

FUNKCJA WBUDOWANA	OPIS
dir([obj])	wraca listę nazw w lokalnym zasięgu (nazwy atrybutów w przypadku obiektu)
locals()	aktualizuje i wraca słownik reprezentujący lokalną tabelę symboli
globals()	wraca słownik reprezentujący bieżącą tablicę globalnych symboli
vars([object])	wraca słownik <code>__dict__</code> podanego obiektu bez argumentów działa jak <code>locals</code>
iter(object[, sentinel])	wraca obiekt iteratora bez drugiego argumentu, podany obiekt musi implementować protokół iteracji (metoda <code>__iter__</code>) lub protokół sekwencji (metoda <code>__getitem__</code>) w przypadku podania obu argumentów, <code>object</code> musi być obiektem, który da się wywołać (bez argumentów) jeśli zwrocona wartość będzie równa <code>sentinel</code> , to zostanie zgłoszony wyjątek <code>StopIteration</code>
next(iterator[, default])	wraca kolejny element z iteratora
open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)	otwiera plik i wraca obiekt pliku

Funkcje wbudowane

FUNKCJA WBUDOWANA	OPIS
id(object)	zwraca wartość typu <i>int</i> unikalną dla tego obiektu i niezmenną w czasie jego życia
hash(object)	zwraca wartość skrótu dla podanego obiektu
range(stop) range(start, stop[, step])	zwraca instancję zakresu
len(s)	zwraca liczbę elementów w kontenerze
filter(function, iterable)	zwraca iterator po elementach <i>iterable</i> dla których warunek <i>function</i> ma wartość <i>True</i>
map(function, iterable, . . .)	zwraca iterator, który stosuje funkcję <i>function</i> do kolejnych elementów <i>iterable</i>
zip(*iterables)	zwraca iterator agregujący elementy z <i>iterables</i>
sorted(iterable, *, key=None, reverse=False)	zwraca nową posortowaną listę elementów <i>iterable</i>
reversed(seq)	zwraca odwrotny iterator
enumerate(iterable, start=0)	zwraca iterator, poprzez który można uzyskać krotki zawierające kolejny indeks oraz element danych

Funkcje wbudowane

FUNKCJA WBUDOWANA	OPIS
super([type[, object-or-type]])	zwraca obiekt proxy, który deleguje wywołania metod do obiektu rodzica
isinstance(object, classinfo)	testuje, czy instancja jest podanego typu lub podtypu
issubclass(object, classinfo)	testuje, czy klasa <i>object</i> jest podklassą <i>classinfo</i>
callable(object)	zwraca <i>True</i> , jeśli obiekt można wywołać jak funkcję (od Pythona 3.2)
help([object])	wywołuje wbudowany system pomocy (do stosowania w trybie interaktywnym)
@classmethod	dekorator konwertujący metodę na metodę klasy
@staticmethod	dekorator konwertujący metodę na metodę statyczną

Biblioteka standardowa

- W ramach dystrybucji Pythona dołączana jest **biblioteka standardowa**
- Wiele **wbudowanych modułów** stanowi integralną część biblioteki standardowej
- Moduły są traktowane jako wbudowane, mimo, że wymagają instrukcji importu
- Oprócz nich można korzystać z modułów dodatkowych, tzw. **rozszerzeń (extensions)**
- Zanim zacznijemy szukać bibliotek zewnętrznych warto upewnić się, czy biblioteka standardowa nie dostarcza rozwiązań naszego problemu

Moduły wbudowane

KATEGORIA	ZAGADNIENIE
dostęp do plików i katalogów	ścieżki do plików, pliki tymczasowe, porównywanie plików → moduły: <i>os</i> , <i>tempfile</i>
matematyka, typy danych	operacje matematyczne, liczby zmiennoprzecinkowe, ułamki, liczby i sekwencje losowe, iteratory → moduły: <i>math</i> , <i>decimal</i> , <i>collections</i>
trwałość danych	serializacja obiektów, baza sqlite, dostęp do baz danych → moduły: <i>pickle</i> , <i>sqlite3</i>
formaty plików	CSV, pliki konfiguracyjne → moduł <i>configparser</i>

Moduły wbudowane

KATEGORIA	ZAGADNIENIE
podstawowe usługi OS	funkcje systemu operacyjnego, czas, argumenty wiersza linii poleceń, dziennik zdarzeń → moduły: <i>os, logging, time, argparse</i>
IPC	komunikacja międzyprocesowa, gniazda → moduły: <i>subprocess, socket</i>
dane internetowe	zarządzanie danymi internetowymi, JSON, poczta elektroniczna, kodowania MIME → moduły: <i>json, email, smtplib, mimetools</i>
strukturalne tagi	parsowanie HTML i XML → moduł: <i>xml.minidom</i>

Moduły wbudowane

KATEGORIA	ZAGADNIENIE
protokoły internetowe	HTTP, FTP, CGI, parsowanie URL, SMTP, POP, IMAP, Telnet, proste serwery → moduły: <i>http</i> , <i>urllib</i> , <i>smtplib</i> , <i>imaplib</i>
rozwój	dokumentowanie, testowanie, konwersja kodu Pythona 2 do 3 → moduły: <i>doctest</i> , <i>2to3</i>
debugowanie	debugowanie, profilowanie → moduły: <i>pdb</i> , <i>profile</i>
runtime	parametry i ustawienia systemowe, elementy wbudowane, ostrzeżenia, konteksty → moduł: <i>inspect</i>

Popularne biblioteki Pythona

NumPy i SciPy

rozbudowane biblioteki matematyczne ułatwiające operacje na dużych zbiorach danych, wykorzystywane w zastosowaniach naukowych

matplotlib

biblioteka graficzna

pandas

biblioteka definiująca struktury danych, ułatwiająca ich analizę

Biblioteki webowe

django

najpopularniejszy framework webowy
posiada własny ORM, upraszczający pracę z bazami danych

pyramid

framework Pythona, początkowo oparty na Pylonie
wspiera aplikacje jednoplikowe, konfigurację opartą na dekoratorach, generowanie URL, itd.

flask

microframework webowy Pythona oparty na *Werkzeug* i *Jinja2*

requests

biblioteka dostarczająca API Pythona do obsługi żądań HTTP
ułatwia pobieranie plików i upraszcza, w porównaniu z biblioteką standardową, pracę z żądaniami HTTP

beautifulsoup

parser HTML umożliwiający nawigację, przeszukiwanie i modyfikację przeparsowanego drzewa dokumentu oraz wyodrębnianie danych ze stron webowych

Inne biblioteki

- Python posiada wiele bibliotek obejmujących różne zastosowania
- Wśród nich można przykładowo wskazać:

Twisted

praca sieciowa

Natural Language Tool Kit (NLTK)

przetwarzanie języka

Pygame

gry w Pythonie

SQLAlchemy

narzędzia bazodanowe

Ćwiczenia/przykłady

- Ćwiczenie/przykład 9.1:
Wieczny kalendarz
- Ćwiczenie/przykład 9.2:
Losowanie lotto
- Ćwiczenie/przykład 9.3:
Kwartaly





Dziękujemy za uwagę