

# PROGRAMOWANIE FUNKCYJNE W PYTHONIE

MODUŁ 4



# Plan szkolenia

## 1 PROGRAMOWANIE FUNKCYJNE

# Plan modułu

## 1 PROGRAMOWANIE FUNKCYJNE

- funkcje – wprowadzenie
- definiowanie funkcji
- parametry funkcji
- funkcje ze zmienną liczbą parametrów
- zasięgi zmiennych i reguła LEGB
- funkcje jako argumenty
- funkcje lambda
- rekurencja
- wzorzec dekoratora
- 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 <b>def</b>
<i>parameters</i>	opcjonalna lista parametrów funkcji

# Definiowanie funkcji

- Funkcja może posiadać dowolną (w tym zerową) liczbę **parametrów**
- 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** – 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

# Parametry *keywords-only*

- Python 3 umożliwia zdefiniowanie parametrów funkcji, które, jeśli funkcja zostanie wywołana, **muszą zostać powiązane z nazwanymi argumentami**
- Takie parametry powinny wystąpić pomiędzy opcjonalnymi deklaracjami parametrów *\*args* i *\*\*kwds*
- Są one określone terminem *keywords-only parameters*
- Jeśli w sygnaturze funkcji nie występuje parametr *\*args*, to do odseparowania parametrów obowiązkowych od parametrów *keywords-only* należy użyć parametru \*
- Parametr \* nie jest wiązany z żadnym z argumentów

## Parametry *positional-only*

- Podczas wywołania funkcji do parametrów pozycyjnych można także odwołać się poprzez nazwę → wtedy kolejność podania argumentów nie jest istotna
- Można także zadeklarować parametry pozycyjne, które **muszą być dostarczone obowiązkowo z pominięciem nazwy**
- Takie parametry noszą nazwę *positional-only parameters*
- Są dostępne od Pythona 3.8
- Do ich odseparowania od parametrów pozycyjnych stosuje się parametr /
- Nie jest on wiązany z żadnym z argumentów

## Ćwiczenia/przykłady



- Ćwiczenie/przykład 1.1:  
*Wieczny kalendarz – użycie funkcji*

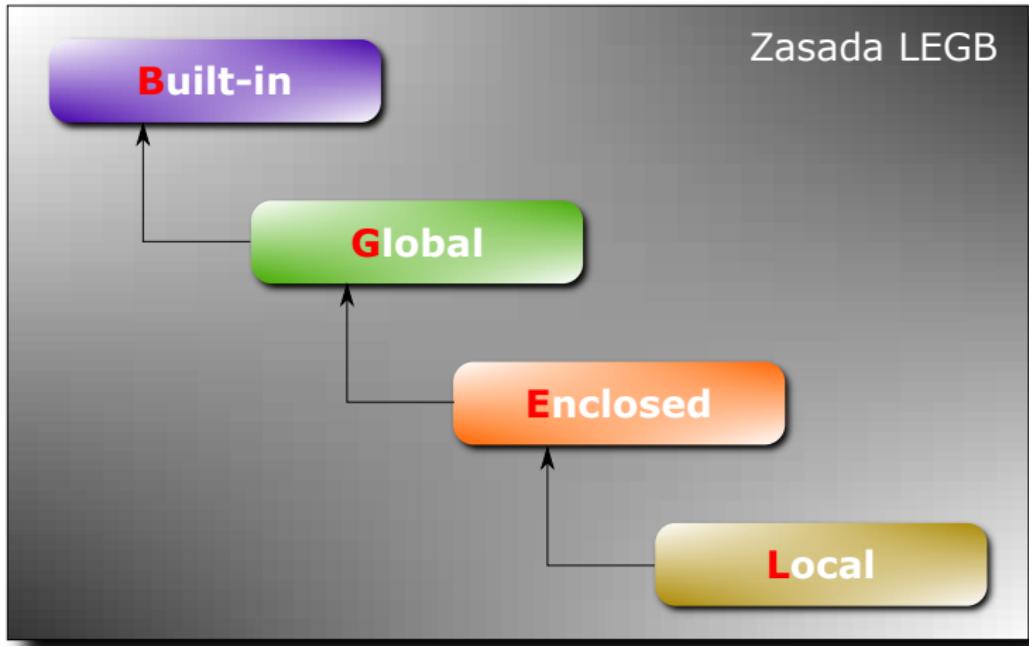
# ĆWICZENIA

# 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

# 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 <b>None</b> 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 1.2:

## **“Baza” osób – funkcje własne i wbudowane**

- Ćwiczenie/przykład 1.3:

## *“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"><li>– pętli (wyrażenie warunkowe jest dopuszczalne)</li><li>– instrukcji <b>return</b></li><li>– instrukcji <b>yield</b></li></ul>

# 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 1.4:  
*“Baza” osób – funkcje lambda*



# Rekurencja

- **Rekurencja (recursion)** to sposób definiowania funkcji, w którym **funkcja wywołuje samą siebie**
- W definicji trzeba pamiętać o umieszczeniu **warunku brzegowego**, który umożliwi zakończenie rekurencji – w przeciwnym razie rekurencja będzie nieskończona
- Zbyt duża liczba wywołań rekurencyjnych mocno obciąża stos i może spowodować jego przepełnienie

## Rekurencja

```
def suma(n):  
    if n == 0:  
        return 0  
    else:  
        return suma(n - 1) + n
```

# Wzorzec dekoratora

- Ponieważ funkcje są obiektami pierwszej klasy, więc można utworzyć funkcję, która:
  - jako argument przyjmie inną funkcję
  - opakuje ją w inną funkcję (*wrapper function*)
  - zwróci nową funkcję
- Ta nowa funkcja może zastąpić oryginalną

# Wzorzec dekoratora

- Funkcję, która zwraca funkcję opakowaną nazywamy **dekoratorem** (*decorator*)

## Funkcja bez dekoratora

```
def powitanie(imie):  
    return f'Cześć {imie}'  
  
komunikat = powitanie('Tomasz')  
print(komunikat)
```

Cześć Tomasz

# Wzorzec dekoratora

## Funkcja “udekorowana”

```
# dekorator
def wersaliki(funkcja):
    print(f'dekorowanie funkcji {funkcja.__name__}')
    def wrapper(*args, **kwargs):
        print(f'wywołanie funkcji {funkcja.__name__}')
        wynik = funkcja(*args, **kwargs)
        return wynik.upper()
    return wrapper

powitanie = wersaliki(powitanie)      # dekorowanie funkcji powitanie

komunikat = powitanie('Tomasz')       # wywołanie funkcji powitanie
print(komunikat)
CZEŚĆ TOMASZ
```

# Wzorzec dekoratora

- Od Pythona 2.4 do deklarowania dekoratorów można użyć alternatywnej składni

Zamiast deklaracji...

```
def funkcja():
    ...
funkcja = dekorator(funkcja)
```

można użyć składni...

```
@dekorator
def funkcja():
    ...
```

# Dekoratory zagnieżdżone

- Dekoratory można zastosować do dowolnej funkcji, w tym także do metod
- Dekoratory można zagnieżdżać:

## Zagnieżdżone dekoratory

```
@a  
@b  
@c  
def funkcja(): ...  
  
# jest równoważne z:  
def funkcja(): ...  
funkcja = a(b(c(funkcja)))
```

- Dekorator trzeba umieścić w linii poprzedzającej dekorowaną funkcję

# Metody fabryki

- Dekoratory mogą przyjmować listę argumentów
- Muszą zwracać funkcję
- W takim przypadku tworzą one **metodę fabryki** (*factory method*)

# 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.

```
>>>
```



Dziękujemy za uwagę