

Python

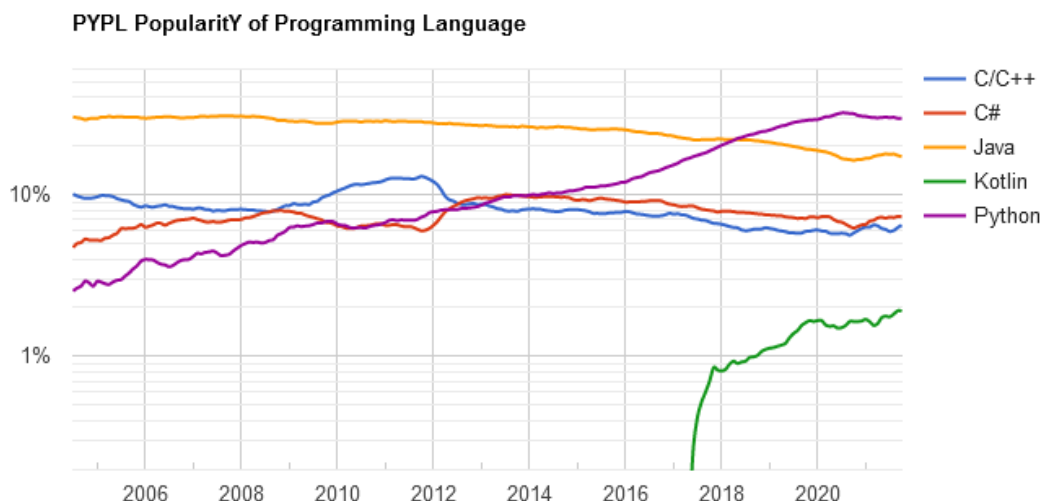
Język programowania Python został stworzony przez holendra Guido van Rossum z CWI (Centrum Matematyki i Informatyki – holenderski narodowy instytut badawczy); Pierwsza wersja powstała w 1990 roku. Nazwa pochodzi od serialu BBC, Monty Python's Flying Circus, z Jonem Cleese. Python jest następcą języka ABC (pierwsza wersja z 1987 r.). Język ABC powstał do nauki programowania – jest to język skryptowy, wysokiego poziomu, interaktywny. ABC i Python są w części inspirowane przez SETL (pierwsze wydanie w 1969 r.), języka bazującego na teorii zbiorów.

Python jest rozwijany jako projekt Open Source przez Python Software Foundation. Podstawowa implementacja, CPython, została napisana w języku C, ale jest też JPython (Java) i IronPython (C#). Obecnie Python jest dostępny w wersji 3.9 (X 2020)

Najważniejsze cechy języka:

- Język skryptowy (interpretowany, nie kompilowany) wysokiego poziomu, ogólnego przeznaczenia – dzięki bogatym bibliotekom
- Z założenia przejrzysty i zwężły, o prostej składni, łatwy do nauki (po ABC – prosta gramatyka, składnia powiązana z formatowaniem, funkcje „uniwersalne”, interaktywny interpreter umożliwiający testowanie kodu, ...)
- Rozszerzalny – ma niewiele funkcjonalności wbudowanych w jądrze, ale za to bogate biblioteki standardowe
- Dynamicznie typowany (typ mają dane, a nie zmienne)
- Wspiera paradygmaty programowania strukturalnego, proceduralnego oraz obiektowego
- Wyposażony w GC (garbage collector)

Python jest obecnie jednym z najbardziej popularnych języków programowania. Istnieją różne rankingi popularności, kolejność zależy od zestawu przyjętych kryteriów, jednak zwykle Python znajduje się w czołówce najbardziej popularnych, ale też najbardziej lubianych języków programowania. (Rysunek 1).



Rysunek 1.1. Popularność wybranych języków programowania.

Źródło: <https://pypl.github.io/>

Filozofia Pythona jest odzwierciedleniem podejścia do pracy i programowania twórców, które wywarło silny wpływ na sam język, ale też jest widoczne w książkach i na forach miłośników. Przykład: „Czytelność kodu jest ważna” – czasami można napisać kod sprytniej, przyspieszając działanie programu o procent lub dwa, ale jeżeli odbywa się to kosztem czytelności, nie powinno się tego robić (jeżeli szybkość jest ważna, można włączyć do Pythona skompilowany kod C/C++). Opisywanie czegoś jako »sprytnego« nie jest uznawane za komplement w kulturze Pythona.

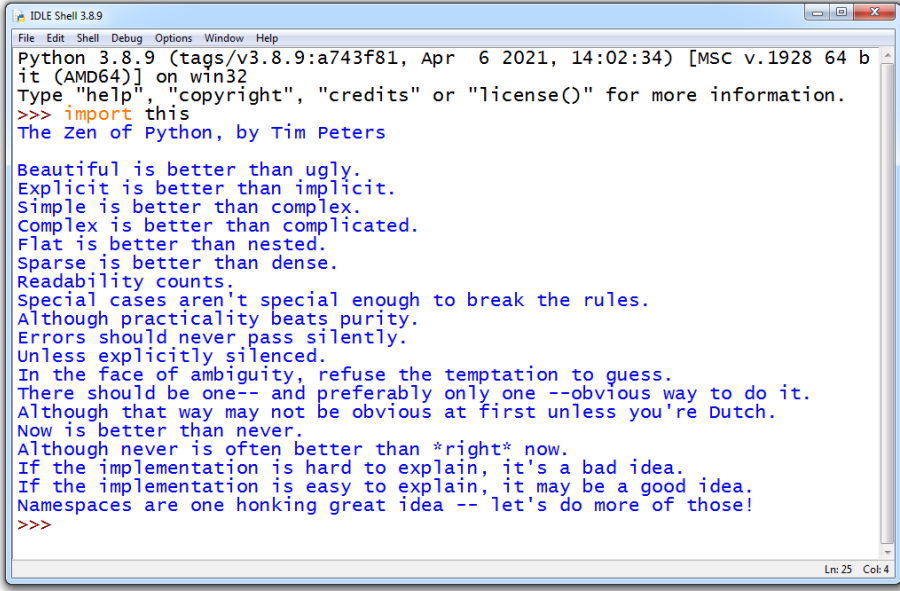
Zen Pythona jest częścią tego języka – gdyby ktoś zapomniał, co w programowaniu jest ważne, wystarczy w interaktywnym interpreterze Pythona napisać (Rysunek 1.2):

```
import this
```

Python w Internecie

W Internecie można znaleźć wiele poradników dotyczących języka Python. Do godnych polecenia z pewnością należą następujące strony internetowe:

- **Python.org** – przewodnik, kompletna specyfikacja języka i biblioteki standardowej, osadzanie C/C++, indeks wszystkich modułów itd. Jest też dokumentacja poprzednich wersji oraz wersja offline do pobrania w różnych formatach; wersja pdf to ~3000 stron <https://docs.python.org/3.9/index.html>
- **Programiz** – przewodnik, przykłady prostych aplikacji, specyfikacja wybranych funkcji, interaktywny interpreter online. Są też podobne przewodniki do innych języków programowania, C++, C#, Java, Kotlin, Javascript. <https://www.programiz.com/python-programming>
- **w3schools** – przewodnik, specyfikacja języka i wybranych modułów, przykłady prostych aplikacji, dodatkowe biblioteki: matematyczne (NumPy, SciPy), bazodanowe (MySQL, MongoDB). Są też podobne przewodniki do języków HTML, CSS i Javascript, platformy Bootstrap i innych technologii związanych z usługą WWW. <https://www.w3schools.com/python/default.asp>



```
IDLE Shell 3.8.9
File Edit Shell Debug Options Window Help
Python 3.8.9 (tags/v3.8.9:a743f81, Apr 6 2021, 14:02:34) [MSC v.1928 64 b
it (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> 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!
>>>
```

Rysunek 1.2 Zen Pythona w środowisku IDLE

Podstawy Pythona

Styl i składnia

Formatowanie jest częścią składni Pythona. Bloki kodu – np. w instrukcji warunkowej lub funkcji – tworzy się przez indentację (tzw. „wcięcie”) zamiast nawiasów klamrowych "{" oraz "}" (C, C++, Java, C#, PHP, ...) lub słów kluczowych "begin" i "end" (Pascal, Delphi). Głębokość wcięcia odpowiada głębokości zagnieżdżenia instrukcji. Wielkość wcięć może się różnić między blokami i między poziomami zagnieżdżenia, ale w jednym bloku musi być stała dla każdego poziomu. Można używać spacji albo tabulacji (ale są one rozróżniane). Zalecana głębokość wcięć to 4 spacje:

```
def fibo(n):  
    a, b = 1, 0  
    for i in range(1, n):  
        a, b = a + b, a  
    return a
```

Łączenie formatowania i składni bywa krytykowane – czy słusznie? Nowoczesne środowiska IDE formatują kod (np. MS Visual Studio), gdyby z kodu usunąć separatory (nawiasy klamrowe i średniki), zostanie kod podobny do Pythona. Jednak te znaki są ważne – dla kompilatora. Zatem są dwa równoległe systemy: separatory, określające strukturę kodu, oraz formatowanie, zapewniające czytelność. Co więcej, można tak sformatować kod, że będzie inaczej rozumiany przez kompilator i inaczej przez człowieka (przy pobieżnym czytaniu). W Pythonie poprawne formatowanie jest częścią składni – kod musi być czytelny, żeby był poprawny

Model danych Pythona

Python traktuje zmienne i wartości w nietypowy sposób. Tradycyjnie, dla typów prostych, zmienna jest miejscem zapisywania wartości (zarezerwowane miejsce w pamięci RAM) – kiedy do zmiennej przypisywana jest nowa wartość, jest umieszczana w tym samym miejscu w pamięci RAM. W Pythonie jest odwrotnie – nowa wartość jest umieszczana gdzieś w pamięci, a zmienna jest etykietką, przypinaną w nowe miejsce. Zmienne o tej samej wartości wskazują to samo miejsce w pamięci, np. w programie Pythona istnieje tylko jeden obiekt „True”. Podobnie zmienne zawierające te same liczby całkowite mają ten sam adres:

```
a = 7  
b = 7  
print(id(a)) # 1865533456  
print(id(b)) # 1865533456
```

Słowa kluczowe

Słowa kluczowe – jak we wszystkich językach programowania wysokiego poziomu – to zbiór słów zastrzeżonych, których nie można "przedefiniować". W Pythonie słów kluczowych jest stosunkowo niewiele i w większości są bardzo często używane

```
False, await, else, import, pass, None, break, except,  
in, raise, True, class, finally, is, return, and,  
continue, for, lambda, try, as, def, from, nonlocal,  
while, assert, del, global, not, with, async, elif, if,  
or, yield
```

Nazwy

Tworząc zmienną, funkcję lub klasę należy nadać im nazwy. Istnieją formalne i nieformalne zasady tworzenia nazw.

Zasady formalne:

- Nazwa może zawierać litery, cyfry oraz znak podkreślenia "_" (tzw. podłoga), można używać znaków diakrytycznych,
- Małe i wielkie litery SA rozróżniane
- Nazwa nie może zaczynać się od cyfry
- Nazwa nie może być słowem kluczowym

Zasady nieformalne:

- Klasy, pola, metody, stałe – zalecana jest konwencja Pacal, przy czym należy stosować jako nazwy klas i pól – rzeczowniki, metod – czasowniki lub wyrażenia czasownikowe:

```
Danewykresu # klasa
KolorLinii   #pole
ZapiszDane   #metoda
```

- Zmienne lokalne i parametry funkcji – konwencja camelCase:

```
promień
polekoła
```

- Nazwy nie powinny zaczynać się od podkreślenia – ze względu na specjalne znaczenie takich nazw – nazwy zaczynające się od podwójnego podkreślenia oznaczają elementy prywatne klas, natomiast nazwy zaczynające i kończące się podwójnym podkreśleniem (tzw. "dunder", double underscore) – oznaczają funkcje specjalne, np. nazwa `__init__` oznacza konstruktor klasy

```
__nazwa      # metody i pola prywatne klas
__nazwa__    # funkcje specjalne
```

Komentarze

Komentarze służą do dokumentowania kodu. Istnieją dwa rodzaje komentarzy:

- Komentarz liniowy – od znaku "#" do końca linii:

```
# Jedna linia wyjaśnienia
```

- Komentarz blokowy – formalnie nie ma, w praktyce stosuje się specjalny wariant łańcucha znaków, ograniczonego potrójnymi cudzysłowami lub apostrofami:

```
"""
    kilka linii
    wyjaśnienia
"""
```

Zmienne

Python jest typowany dynamicznie, co oznacza że zmiennych się nie deklaruje (więc i nie określa typu). Typ zmiennej jest ściśle określony, na podstawie wartości, jaka została przypisana zmiennej. Typ zmiennej może się zmienić, jeżeli do tej samej zmiennej zostanie wstawiona wartość innego typu. Typ można sprawdzić, przy pomocy funkcji `type()`

Typy prosté

Python ma niewielką liczbę wbudowanych typów prostych. Należą do nich:

- bool

Wartości logiczne, True i False (są to słowa kluczowe)

p = True

- `int`

Liczby całkowite ze znakiem, zakres nieograniczony

```
i = 100000000000000000000000000000000000000000000000000000
```

- float

Liczby rzeczywiste, zakres wartości (w przybliżeniu) $0.0, \pm (1e-300 \dots 1e300)$. Po przekroczeniu tego zakresu przyjmuje wartości `inf` lub `-inf` albo `nan` (jak zmienne typu `double` w C#); Do wykrycia tego można użyć funkcji modułu `math`, `isinf` oraz `isnan()`

- complex

Liczby zespolone, przy czym składowe rzeczywista (real) i urojona (imag), zawsze są typu float i są właściwościami tylko do odczytu;

```
i = 1000**100 # int
x = 13.0       # float
y = 1.3e1      # float
z = 1 + 1j     # complex, z.real i z.imag tylko do odczytu
```

- str

Łańcuch znaków, ograniczony apostrofami lub cudzysłowami; Długie łańcuchy (obejmujące więcej niż jedną linię) – należy ograniczyć przez trzy znaki " lub ';

Wewnętrznie Python używa Unicode do zapisu znaków

```
s = "Hello, world!"
```

Znaki specjalne zapisuje się przy użyciu sekwencji ucieczki, np. `\n`, `\t`, `\"` – aby to wyłączyć, należy użyć trybu „raw”:

```
r = r"Hello, \n world!" # \n, a nie nowa linia
```

Można używać operatora indeksowania, ale tylko do odczytu (łańcuchy są „immutable” – niemodyfikowalne)

- None

Jedyna wartość typu `NoneType`, oznacza brak wartości; Zmienna, której zostanie przypisana wartość `None`, istnieje; może być testowana operatorami równości

Konwersje typów prostych

Konwersja automatyczna typów prostych w Pythonie jest wykonywana, ale zdarza się rzadziej, niż np. w C# – ponieważ w Pythonie, kiedy do zmiennej wstawiana jest wartość innego typu, to zmienia się typ zmiennej, a nie konwertuje wartość. Konwersja automatyczna może być wykonywana przy wykonywaniu operatorów lub przy wywołaniu funkcji.

Konwersje automatyczne mogą być wykonywane pomiędzy następującymi typami:

- z int na float
- z float na complex
- z int, float, complex, str na bool

Konwersje wymuszone działają pomiędzy wszystkimi typami prostymi, w tym z typu string na typy liczbowe, co jest bardzo wygodne. Jest to dość typowe dla języków skryptowych, ze względu na dynamiczne typowanie, jednak w przeciwieństwie do np. języka PHP, Python rzuci wyjątek programowy kiedy konwersja skończy się niepowodzeniem.

```
complex("2+3j")      # ok
float("1.234e56")     # ok
int("dwa")            # błąd, wyjątek programowy
bool(13)              # ok
int(True)             # ok, 1
str(False)            # ok, "False"
bool("False")         # ok, True (!)
```

Moduły i przestrzenie nazw

Każdy plik Pythona może być jednocześnie programem i modulem innego programu. Ułatwia to pracę nad modułami – można je wygodnie sprawdzać. Plik uruchamiany jako program jest umieszczany w głównej przestrzeni nazw ("__main__"), co można sprawdzić instrukcją warunkową. Instrukcje poza taką instrukcją warunkową zostaną wykonane podczas importowania modułu, jako jego inicjalizacja

```
# moduł – klasy i funkcje udostępniane
def foo():
    pass

# testowanie modułu
if __name__ == '__main__':
    foo()
```

Nazwa jest identyfikatorem zmiennej, funkcji lub klasy. Przestrzeń nazw to zbiór nazw wszystkich obiektów jednego modułu, przy czym w programie może współistnieć wiele przestrzeni nazw. Przestrzenie nazw pozwalają unikać konfliktów nazw bibliotek różnego pochodzenia. W Pythonie (inaczej niż w językach C++, Java, C#) z zasady każdy plik to oddzielna przestrzeń nazw. Jeżeli moduły podzielone są na katalogi, to nazwa katalogu jest częścią nazwy przestrzeni nazw (chyba że pliki z danego katalogu zostaną połączone w pakiet, o czym świadczy obecność pliku __init__.py w tym katalogu).

Program może korzystać z modułu dzięki dyrektywie import, na dwa sposoby:

- Importowanie całej przestrzeni nazw, z zachowaniem jej odrębności

```
import math
x = math.sin(math.pi)
```

- Importowanie wybranej nazwy z przestrzeni nazw (wówczas nazwa zostanie włączona do bieżącej przestrzeni nazw)

```
from math import cos
x = cos(0.0)
```

Można użyć obu sposobów jednocześnie, formalnie nie jest to błędem. Powtórzenie instrukcji importu nie jest błędem – jest ignorowane

Operatory i wyrażenia

Python ma dość unikalny zestaw operatorów, czym wyróżnia się spośród innych popularnych języków programowania. Również hierarchia operatorów jest nietypowa, co należy uwzględnić przy tworzeniu wyrażeń:

- Indeksowanie, wycinanie (slicing), wywołanie funkcji, odwołanie do atrybutu
`x[i]` `x[i:j]` `f()` `x.a`
- Potęgowanie
`**`
- Jednoargumentowe
`+x` `-x` `~x`
- Mnożenia i dodawania
`*` `/` `//` `%`
`+` `-`
- Przesuwania bitów
`<<` `>>`
- Bitowe
`&`
`^`
`|`
- Relacji, równości, identyczności, zawierania
`<` `<=` `>` `>=` `==` `!=`
`is` `is not`
`in` `not in`
- Logiczne
`not`
`and`
`or`

Operatory arytmetyczne

Do operatorów arytmetycznych należą:

```
** - potęgowanie
*  - mnożenie
/  - dzielenie
// - dzielenie z zaokrągleniem w dół
%  - reszta z dzielenia
+  - dodawanie
-  - odejmowanie
```

Operator potęgowania jest dwa szczeble wyżej w hierarchii, niż operatory mnożenia, operatory dodawania o szczebel niżej. Typ rezultatu zależy od operatora oraz typu operandów (argumentów). Na ogół jeżeli oba operandy są `int`, to wynik jest `int`, jeżeli chociaż jeden jest `float`, to wynik jest `float`, jeżeli chociaż jeden jest `complex`, to wynik jest `complex`. Wynik dzielenia `/` zawsze jest `float` lub `complex`. Dzielnik `//` zawsze zaokrągla w dół (np. dzielenie `3//2` daje wynik `1`, natomiast `-3//2` daje `-2`), zatem nie jest to odrzucenie części ułamkowej, jak w językach C/C++/Java (np. `int(-3/2)` daje wynik `-1`)

Operatory relacji, równości i tożsamości

Oprócz typowych operatorów relacji i równości, Python ma również operatory tożsamości i zawierania:

```
<, <=, >, >= - relacji
==, !=       - równości
```

`is, is not` - tożsamości
`in, not in` - zawierania

Wszystkie te operatory są imperatywne (dają odpowiedź na pytanie „czy to prawda, że ...”), o rezultacie boolowskim. Wszystkie mają tę samą hierarchię – fakt ten ma jednak znikome znaczenie, ponieważ prawie nigdy nie są łączone kaskadowo:

```
((x==y) >= (x==z)) is not (y>z)
```

Operatory te są w hierarchii poniżej arytmetycznych i powyżej logicznych, zatem ich naturalne łączenie nie wymaga nawiasów, co zwiększa czytelność, np. tu najpierw zostaną wykonane mnożenia, potem relacje, na koniec koniunkcja

```
x > 0.5*y and x <= 2*y
```

Operatory relacji można łączyć w sekwencje, np.:

```
x < y < z
```

przy czym nie jest stosowane wiązanie, np. $(x < y) < z$, jest to tylko skrócenie zapisu: powyższe wyrażenie jest równoważne następującemu:

```
(x < y) and (y < z)
```

Relacja pomiędzy pierwszym a trzecim operandem nie jest sprawdzana, zatem wyrażenie poniżej, chociaż nie jest eleganckie, jest poprawne:

```
x < y > z    # równoważnie: y>x and y>z
```

Operatory tożsamości porównują identyfikatory ("adresy") obiektów. Są przeznaczone do sprawdzania, czy zmienne referencyjne wskazują ten sam obiekt. Wbudowane typy niezmiennicze ("immutable", w tym łańcuchy i krotki) o tej samej wartości są tożsame, podobnie jak stałe `True`, `False` i `None`; wbudowane typy zmienne (w tym listy) oraz obiekty, nawet o tej samej wartości – nie są tożsame

```
(1==1) is (2==2)    # True  
"to be" is "to be"   # True  
(2,3) is (2,3)      # True (krotka, kolekcja immutable)  
[2,3] is [2,3]      # False (lista, kolekcja mutable)
```

Operatory zawierania stosuje się do kolekcji, a w tym do łańcuchów

```
2 in (1, 2, 3)                    # True  
  
colors = ["red", "blue"]  
"red" in colors                  # True  
  
"D" in "Django"                 # True
```

Operatory logiczne

Operatory logiczne są zapisywane słownie, w odróżnieniu od operatorów bitowych (znów, inaczej niż w C/C++/Java), co zwiększa czytelność kodu:

not - negacja
and - koniunkcja
or - alternatywa

Operator negacji jest w hierarchii niżej, niż inne operatory jednoargumentowe, nawet poniżej operatorów relacji (po raz kolejny inaczej, niż C/C++/Java/C#)

```
not 2*y > 3    # nawiasy zbędne, not i tak ostatnie
```

W Pythonie argumentami operacji logicznych mogą być dowolne typy proste, chociaż oczywiście bardziej przejrzyste jest użycie wyników relacji. Wszystkie typy proste są automatycznie konwertowane na wartość boolowską według zasady: None, 0, 0.0, 0+0j oraz "" (pusty łańcuch) dają False, wszystkie inne wartości – True

Python optymalizuje obliczanie koniunkcji i alternatywy: jeżeli w wyrażeniu (a and b) czynnik a ma wartość False, to całość będzie False, niezależnie od wartości b – zatem b nie jest wyznaczane, zaś wyrażenie zwraca czynnik a (a nie False) jako wynik.
Np. jeżeli Link.IsOpen jest false, to ReadData() nie zostanie wywołane:

```
ReadOK = Link.IsOpen and Link.ReadData();
```

Czasami ta właściwość jest wykorzystywana dla skrócenia kodu, ponieważ odpowiednio użyta zastępuje instrukcję if-else:

```
if not Link.IsOpen:  
    ReadOK = false  
else:  
    ReadOK = Link.ReadData()
```

Połączenie zasad automatycznej konwersji oraz optymalizacji może dawać zaskakujące rezultaty, np.:

```
"to be" or "not to be"    # "to be" (sic!)
```

niepusty łańcuch jest równoważny True, zatem drugi operand nie jest brany pod uwagę i zwracana jest wartość pierwszego operandu – jednak bez konwertowania go na wartość boolowską

Operator i instrukcja przypisania

W Pythonie występuje operator przypisania oraz kilka rozszerzonych operatorów przypisania:

```
=  
**= *= /= // = %= += -=
```

Operatory przypisania nie występują w tabelce hierarchii, ponieważ nie są traktowane jak inne operatory, tworzą instrukcję przypisania:

```
<zmienna> = <wyrażenie>
```

oznacza to, że nie można użyć przypisania np. tak:

```
x = 2 * (y = 3)    # błąd!
```

Można użyć jedno- albo wielokrotnego przypisania:

```
x = 13
x = y = z = 7
```

jednak w wielokrotnym przypisaniu nie można użyć rozszerzonych operatorów przypisania:

```
x *= y = 7    # błąd, jak dla x = x * y = 7
```

Specjalne traktowanie przypisania pozwala uniknąć błędów wynikających z mylenia operatorów równości i przypisania:

```
if (x == y):    # ok
    print('równe')

if (x = y):     # błąd!
    print('równe')
```

Specyficzną cechą przypisania w Pythonie (i niektórych innych językach skryptowych) jest automatyczne rozpakowanie kolekcji, co umożliwia przypisanie wartości kilku zmiennym w jednej instrukcji:

```
(a, b) = (x, y)
```

Można to wykorzystać np. do zwracania przez funkcję wielu wartości:

```
def f(x):
    return (x-1, x+1)

xLess, xMore = f(13)    # xLess=12, xMore=14
x = f(13)               # x jest krotką, x = (12, 14)
```

Ten mechanizm umożliwia też zamianę wartości zmiennych przy użyciu jednej tylko instrukcji (nawiasy są opcjonalne):

```
(a, b) = (b, a)
```

Wyrażenie przypisania

Wyrażenie przypisania, wprowadzone do języka w jego najnowszej wersji (3.9), wykorzystuje specjalną wersję operatora przypisania:

```
:=
```

Wyrażenie przypisania może wystąpić tam, gdzie nie można użyć instrukcji przypisania (i na odwrót), np.:

```
while len(data := readData()) > 0:
    # dostępna kolekcja data, rozmiar niezerowy

if (y := f(x)) > 0:
    # dostępna zmienna y, wartość dodatnia

y = [ a := f(x), a**2, a**3 ]
```

Daje możliwość przypisania w wyrażeniach (jak w językach C++/C#), jednak bez łamania wcześniejszych zasad dotyczących instrukcji przypisania.

Wyrażenie warunkowe

Wyrażenie warunkowe ma postać:

```
<w1> if <warunek> else <w2>
```

Jest to odpowiednik wyrażenia warunkowego ?: z języków C++/Java/C#, jednak bez wprowadzania nowego operatora. Pozwala skrócić zapis prostej instrukcji warunkowej:

```
y = 1.0 if x==0.0 else sin(x)/x
```

```
if x = 1.0:
    y = 1.0
else:
    y = sin(x)/x
```

Ze względu na dynamiczne typowanie, nie ma ograniczeń języków C-podobnych, np. wyrażenia <w1> i <w2> nie muszą być tego samego typu, jak w poniższych przykładach:

```
y = e**x if x>=0 else None
x = (x1, x2) if delta>=0 else "brak pierwiastków"
```

Korzystając z wyrażenia warunkowego w taki sposób należy zachować szczególną staranność, ponieważ zmienność typu rezultatu może prowadzić do błędów w dalszej części kodu.

Instrukcje

Python dysponuje mniejszą liczbą instrukcji strukturalnych, zwłaszcza instrukcji iteracyjnych, niż inne języki programowania. Należą do nich:

- Instrukcja pusta
- Instrukcja wyrażeniowa
- Instrukcja warunkowa (if-elif-else)
- Instrukcje iteracyjne (for, while)

Instrukcja pusta

Składnia:

```
pass
```

Instrukcja pusta może być użyta, kiedy składnia wymaga instrukcji, a nic nie powinno być wykonane, np.:

```
if <warunek>:
    pass
```

```
def f():
    pass
```

```
class c:
    pass
```

Instrukcja pusta jest często wykorzystywana do budowania "szkieletów" klas i funkcji, które dzięki niej są poprawne składniowo, a później mogą być uzupełnione.

Instrukcja wyrażeniowa

Składnia:

```
<wyrażenie>
```

Wyrażenie to poprawna składniowo kombinacja stałych, zmiennych, operatorów i wywołań funkcji, np.:

```
a = b = c = 13
x += 7
f(x)
y = 3 + f(x)
y = 1.0 if x == 0.0 else sin(x)/x
a + b
x
```

Ograniczenia:

- cała instrukcja powinna się mieścić w jednej linii
- w jednej linii powinna być użyta jedna instrukcja
- przypisania mogą być łączone w sekwencje, ale rozszerzone przypisania już nie

Wyrażenia bez przypisania są legalne, ponieważ mogą być przydatne w trybie interaktywnym

Jeżeli instrukcja nie mieści się w jednej linii, należy użyć znaku "\"

```
suma = a + b + c + \
        d + e + f
```

Jeżeli częścią instrukcji są nawiasy, to znak "\"" jest opcjonalny:

```
lista = ( 1, 2, 3,
          4, 5, 6 )
```

W obu powyższych przypadkach spacje wiodące w linii z kontynuacją instrukcji nie są częścią składni, ich liczba jest dowolna

Jeżeli w jednej linii ma być zapisanych kilka instrukcji, należy je oddzielić średnikami (średnik na końcu linii z jedną instrukcją nie jest błędem):

```
x = 1; y = 2;
```

Instrukcja warunkowa

Składnia:

```
if <warunek1>:
    <instrukcja1>
elif <warunek2>:
    <instrukcja2>
else:
    <instrukcja3>
```

Instrukcja musi zawierać sekcję `if`, może zawierać dowolną liczbę sekcji `elif` oraz opcjonalnie jedną sekcję `else`, która – o ile jest użyta – musi być na samym końcu. Jako warunek można użyć dowolnego wyrażenia o rezultacie boolowskim albo mającego automatyczną konwersję na typ `bool`; Dla typów prostych: `0`, `0.0`, `0.0+0.0j`, `""` jest konwertowane na `False`, wartości niezerowe oraz niepusty łańcuch – na `True`, zatem np. dla typów liczbowych `if x:` jest równoważne `if x != 0:`

Przykład:

```

if x < 0:
    print("poniżej 0")
elif 0 <= x < 5:
    print("między 0 a 5")
else:
    print("powyżej 5")

```

Instrukcja iteracyjna for

Składnia:

```

for <zmienna> in <kolekcja lub generator>:
    <instrukcja>

```

Wykonanie instrukcji for polega na przypisaniu do zmiennej kolejnych wartości pobieranych z kolekcji (albo z generatora) i wykonywaniu – dla każdej z tych wartości – instrukcji iterowanej. Kolekcja może być dowolnego typu (lista, krotka, zbiór, ...). Istnieją generatory wbudowane, np. range, ale można też definiować własne

Przykład:

```

kolory = ("czerwony", "niebieski")
for kolor in kolory:
    print(kolor)

```

```

liczby = [1, 3, 5, 7, 11, 13]
for liczba in liczby:
    print(liczba)

```

Generator range

Generator range umożliwia utworzenie bardziej klasycznej pętli z licznikiem:

```

for i in range(10):
    print(i, i**2)

f1 = 1
f2 = 1
for i in range(3, liczba+1):
    f2, f1 = f1+f2, f2

fibonacci = f2

```

Generator range ma trzy warianty:

- range(stop) – od 0 do stop-1, co 1
- range(start, stop) – od start do stop-1, co 1
- range(start, stop, step) – od start do stop-1, co step

Instrukcja iteracyjna while

Składnia:

```

while <warunek>:
    instrukcja

```

Zastosowanie i sposób wykonania – jak w językach C-podobnych. Jako warunek można użyć dowolnego wyrażenia o rezultacie boolowskim albo mającego automatyczną konwersję na typ

bool; podobnie jak w przypadku instrukcji warunkowej, zatem np. dla typów liczbowych `while x:` jest równoważne `while x != 0:`

Przykład:

```
plik = open("plik.txt")

while (line := plik.readline()) != '':
    print(line, end='')

plik.close()
```

Instrukcje `break` i `continue`

Instrukcje `break` i `continue` służą do kontroli wykonania instrukcji iteracyjnych – do przerywania wykonywania instrukcji iteracyjnej (`break`) albo bieżącej iteracji (`continue`); Używa się ich wyłącznie w połączeniu z instrukcją warunkową

```
kolory = ("czerwony", "zielony", "niebieski")

for kolor in kolory:
    if 'r' in kolor:
        continue
    print(kolor)
```

Wejście i wyjście konsoli

Standardowe wejście i wyjście aplikacji konsolowych tworzą funkcje `print` oraz `input`.

Funkcja `input` ma postać:

```
input(prompt='')
```

Funkcja ta wyświetla prompt, wczytuje znaki wpisywane z klawiatury aż do naciśnięcia klawisza <Enter> i zwraca wczytany łańcuch (ale bez znaku końca linii), np.:

```
liczba = float(input("Podaj liczbę: "))
```

Funkcja `print`:

```
print(*objects, sep=' ', end='\n')
```

Funkcja wyświetla argumenty `*objects` (może być dowolna liczba argumentów, rozdzielonych przenikami) przekonwertowane na łańcuchy znaków, rozdzielając je sekwencją `sep` (domyślnie spacja) i dodaje na końcu sekwencję `end` (domyślnie znak końca linii);

```
print(a, b, c, sep='\n', end='')
```

Formatowanie łańcuchów znaków

Python oferuje kilka metod formatowania łańcuchów znaków. Z biegiem czasu wprowadzano nowsze, wygodniejsze i bardziej wydajne, jednak dla zachowania kompatybilności wstecz, starsze metody nie zostały wycofane.

Zalecane jest użycie funkcji `str.format()` lub f-string, chociaż dostępne jest też formatowanie starszą metodą, przy użyciu operatora `%`, np.:

```
'a = %f' % a
```

Metoda format jest dostępna zarówno przez klasę str, jak i przez jej obiekty – czyli łańcuchy znaków:

```
str.format("formatowanie", wartość)  
"formatowanie".format(wartość)
```

Formatowanie może zawierać miejsce na wartości, w postaci:

- {} – wartości są wstawiane według kolejności wystąpienia
- {0} {1} – wartości są wstawiane według kolejności indeksów
- {nazwa} – wartości są wstawiane według nazw argumentów

```
str.format("suma {} oraz {} wynosi {}", a, b, a+b)  
str.format("{2} jest sumą {0} oraz {1}", a, b, a+b)  
str.format("{s1} + {s2} = {s}", s1=a, s2=b, s=a+b)
```

Można też korzystać z interpolacji łańcuchów (potocznie znane jako f-string), co pozwala włączać do łańcucha wartości zmiennych

```
zmienna = ...  
f'dowolny tekst, {zmienna}'
```

W bardziej złożonych przypadkach, np. lokalizacji całej aplikacji, albo wielokrotnego używania określonego wzorca wydruku, można wykorzystać szablony łańcuchów (template strings), np.:

```
welcomeTemplate = Template("Hello, $name")  
  
userName = ...  
welcome = welcomeTemplate.substitute(name=userName)
```

Zadania

Proszę napisać program, który...

1. Wyświetla napis "Hello world!"
Program należy uruchomić ze środowiska IDLE oraz z linii poleceń
2. Oblicza i wyświetla pierwiastki równania kwadratowego, dla danych (a, b, c):
(1) 2, -10, 12, (2) 1, 2, 1, (3) 1, 2, 2
3. Jak w punkcie 2, ale wykonuje obliczenia na liczbach zespolonych
4. Jak w punkcie 2, ale wykonuje obliczenia na liczbach rzeczywistych dla $\Delta \geq 0$ oraz na liczbach zespolonych dla $\Delta < 0$
5. Drukuje liczby od 1 do 10 oraz ich kwadraty i sześciany
6. Drukuje tabelkę wartości funkcji sinus i cosinus, dla kątów 0, 10, 20, ..., 90 stopni
7. Drukuje tabelkę kolejnych potęg liczby 2, tj. n i 2^n , dla $n=1, 2, \dots, 10$
8. Drukuje tabelkę funkcji $\sin(x)/x$, dla x = od 0 do 3π , co $\pi/4$
9. Jak w punkcie 8, ale z użyciem wyrażenia warunkowego zamiast instrukcji warunkowej
10. Drukuje pierwszych 20 liczb ciągu Fibonacciego $F(1)=1, F(2)=1, F(3)=3, \dots$
11. Jak w punkcie 10, ale z użyciem rozpakowania kolekcji $(a, b) = (c, d)$
12. Wyznacza i drukuje n -ty element ciągu Fibonacciego
13. Jak w dwóch w punktach 6-8, ale z użyciem pętli while
14. Wyznacza NWP metodą Euklidesa, z użyciem odejmowania
15. Jak w punkcie 14, ale z użyciem rozpakowania kolekcji $(a, b) = (c, d)$
16. Jak w punkcie 14, ale z użyciem reszty z dzielenia
17. Jak w punkcie 16, ale z użyciem rozpakowania kolekcji $(a, b) = (c, d)$
18. Wyznacza przybliżoną wartość pierwiastka kwadratowego z zadaną dokładnością, z użyciem funkcji fabs()
19. Jak w punkcie 18, ale z użyciem łączenia operatorów relacji $a < b < c$
20. Jak w punkcie 18, ale z użyciem pętli for