Poniższy rozdział pochodzi z książki:

Charles R. Severance - "Python dla wszystkich: Odkrywanie danych z Python 3"

Pełna wersja podręcznika znajduje się na stronie https://py4e.pl/book

Rozdział 3

Wykonanie warunkowe

3.1. Wyrażenia logiczne

Wyrażenie logiczne to wyrażenie, które jest prawdziwe lub fałszywe. Poniższe przykłady używają operatora ==, który porównuje dwa operandy i zwraca `True`, jeśli są one równe, lub `False` w przeciwnym wypadku:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

`True` i `False` są specjalnymi wartościami, które należą do klasy `bool`; nie są one napisami:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

Operator == jest jednym z *operatorów porównania*; pozostałe to:

```
x != y               # x nie jest równe y
x > y                # x jest większe od y
x < y                # x jest mniejsze od y
x >= y               # x jest większe lub równe y
x <= y               # x jest mniejsze lub równe y
x is y               # x jest tym samym co y
x is not y           # x nie jest tym samym co y
```

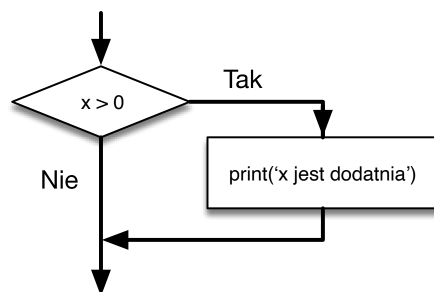
Choć powyższe operacje pewnie już znasz, symbole używane przez Pythona różnią się od symboli matematycznych. Częstym błędem jest użycie pojedynczego znaku równości (=) zamiast znaku podwójnej równości (==). Pamiętaj, że = jest operatorem przypisania, a == jest operatorem porównania. Nie ma znaków takich jak =< lub =>.

3.2. Operatory logiczne

Istnieją trzy *operatory logiczne*: `and`, `or` i `not`. Semantyka (znaczenie) tych operatorów jest podobna do ich znaczenia w języku angielskim. Na przykład wyrażenie

`x > 0 and x < 10`

jest prawdziwe tylko wtedy, gdy wartość zmiennej `x` jest większa niż 0 *i* mniejsza niż 10.



Rysunek 3.1. Logika instrukcji if

Wyrażenie `n%2 == 0 or n%3 == 0` jest prawdziwe, jeśli *którykolwiek* z tych warunków jest prawdziwy, tzn. jeśli liczba jest podzielna przez 2 *lub* 3.

Operator `not` neguje wyrażenie logiczne, więc `not (x > y)` jest prawdziwe, jeśli `x > y` jest fałszywe, tzn. jeśli wartość `x` jest mniejsza lub równa `y`.

Ściślej mówiąc, operandy operatorów logicznych powinny być wyrażeniami logicznymi, ale Python nie jest tutaj bardzo restrykcyjny. Każda dodatnia liczba całkowita jest interpretowana jako "prawdziwa".

```
>>> 17 and True
True
```

Taka elastyczność może być użyteczna, ale istnieją pewne niuanse, które mogą być mylące. Lepiej ich unikać, dopóki nie będziesz pewien, że wiesz, co robisz.

3.3. Instrukcja warunkowa

Aby napisać użyteczny program, prawie zawsze potrzebujemy możliwości sprawdzenia pewnych warunków i dostosowania do nich zachowania programu. *Instrukcje warunkowe* dają nam tę możliwość. Najprostszą formą jest instrukcja `if`:

```
if x > 0 :
    print('x jest dodatnia')
```

Wyrażenie logiczne po instrukcji `if` jest nazywane *warunkiem*. Kończymy wyrażenie `if` znakiem dwukropka (`:`), a linia lub linie po wyrażeniu `if` są wcięte.

Jeśli warunek logiczny jest prawdziwy, to wykonywane jest wyrażenie we wcięciu. Jeśli warunek logiczny jest fałszywy, to wyrażenie we wcięciu jest pomijane.

Instrukcje `if` mają taką samą strukturę jak definicje funkcji lub pętle `for`[1]. Instrukcja składa się z linii nagłówka, która kończy się znakiem dwukropka (`:`), po którym następuje wcięty blok (tzw. *ciało* instrukcji `if`). Takie instrukcje nazywane są *instrukcjami złożonymi*, ponieważ rozciągają się na więcej niż jedną linię.

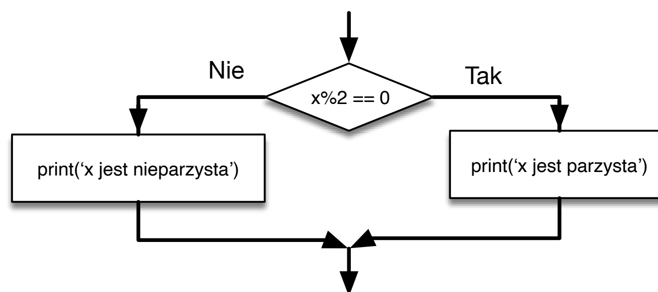
Nie ma górnego ograniczenia co do liczby instrukcji, które mogą pojawić się w ciele instrukcji `if`, ale musi być co najmniej jedna instrukcja. Czasami warto przygotować ciało `if` bez instrukcji (zazwyczaj jako miejsce na kod, którego jeszcze nie napisałeś). W takim przypadku możesz użyć `pass`, który nie robi nic.

```
if x < 0 :
    pass          # trzeba obsłużyć wartości ujemne!
```

Jeśli wpiszesz instrukcję `if` w interpreterze Pythona, znak zachęty zmieni się z jodełki na trzy kropki, tak aby wskazać, że jesteś w środku bloku instrukcji:

```
>>> x = 3
```

[1] Dowiemy się o funkcjach w rozdziale 4, a o pętlach w rozdziale 5.



Rysunek 3.2. Logika if-else

```
>>> if x < 10:
...     print('malutko')
...
malutko
>>>
```

Kiedy używasz interpretera Pythona, musisz zostawić pustą linię na końcu bloku, bo w przeciwnym razie Python zwróci błąd:

```
>>> x = 3
>>> if x < 10:
...     print('malutko')
... print('zrobione')
  File "<stdin>", line 3
    print('zrobione')
        ^
SyntaxError: invalid syntax
```

Pusta linia na końcu bloku instrukcji nie jest konieczna podczas pisania i wykonywania *skryptu*, ale może poprawić czytelność Twojego kodu.

3.4. Wykonanie alternatywnego bloku kodu

Drugą formą instrukcji `if` jest *wykonanie alternatywne*, w którym istnieją dwie możliwości i to warunek określa, która z nich zostanie wykonana. Składnia wygląda następująco:

```
if x%2 == 0 :
    print('x jest parzysta')
else :
    print('x jest nieparzysta')
```

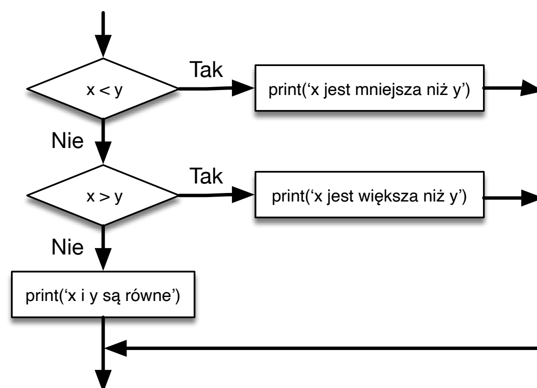
Jeśli reszta z dzielenia `x` przez 2 jest równa 0, to wiemy, że zmienna `x` jest parzysta, a program wyświetli stosowny komunikat. Jeśli warunek jest fałszywy, to wykonywany jest drugi zestaw poleceń.

Ponieważ warunek musi być albo prawdziwy, albo fałszywy, dokładnie jedna z alternatyw zostanie wykonana. Te alternatywy nazywane są *gałęziami* w przepływie wykonania programu.

3.5. Warunki powiązane

Czasami mamy więcej niż dwie możliwości i potrzebujemy więcej niż dwóch gałęzi. Jednym ze sposobów na wyrażenie takiej sytuacji jest *warunek powiązany*:

```
if x < y:
    print('x jest mniejsza niż y')
```



Rysunek 3.3. Logika if-elif

```
elif x > y:
    print('x jest większa niż y')
else:
    print('x i y są równe')
```

`elif` jest skrótem od *else if*. Tutaj również wykonana zostanie dokładnie jedna gałąź.

Nie ma ograniczenia co do liczby instrukcji `elif`. Jeśli istnieje klauzula `else`, to musi ona być na końcu (ale samo jej pojawienie się nie jest wymagane).

```
if choice == 'a':
    print('Zła odpowiedź')
elif choice == 'b':
    print('Dobra odpowiedź')
elif choice == 'c':
    print('Blisko, ale źle')
```

Każdy warunek jest kolejno sprawdzany. Jeśli pierwszy jest fałszywy, sprawdzany jest następny itd. Jeśli jeden z nich jest prawdziwy, to wykonywana jest odpowiadająca mu gałąź, a instrukcja jest kończona. Nawet jeśli więcej niż jeden warunek jest prawdziwy, to tylko pierwszy z tych prawdziwych wykonuje swoją gałąź kodu.

3.6. Warunki zagnieżdżone

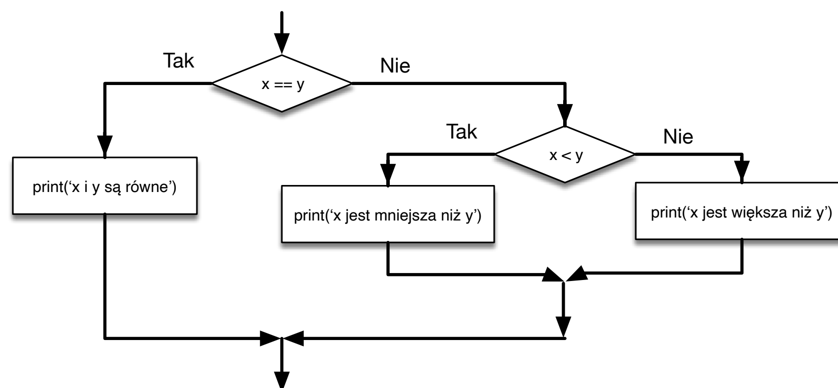
Jeden warunek może być również zagnieżdżony w innym. Mogliśmy zapisać przykład z trzema gałęziami w ten sposób:

```
if x == y:
    print('x i y są równe')
else:
    if x < y:
        print('x jest mniejsza niż y')
    else:
        print('x jest większa niż y')
```

Warunek zewnętrzny zawiera dwie gałęzie. Pierwsza gałąź zawiera prostą instrukcję. Druga gałąź zawiera kolejną instrukcję `if`, która ma dwie własne gałęzie. Te dwie gałęzie to proste instrukcje, chociaż mogłyby być również instrukcjami warunkowymi.

Chociaż wcięcie tych instrukcji sprawia, że ich struktura jest widoczna, to jednak *zagnieżdżone instrukcje warunkowe* bardzo szybko stają się trudne do odczytania. Ogólnie rzecz biorąc, jeśli możesz, to ich unikaj.

Operatory logiczne często są sposobem na uproszczenie zagnieżdżonych instrukcji warunkowych. Na przykład możemy przepisać następujący kod za pomocą jednego warunku:



Rysunek 3.4. Zagnieżdżone instrukcje if

```
if 0 < x:
    if x < 10:
        print('x jest jednocyfrową dodatnią liczbą.')
```

Funkcja `print()` jest wywoływana tylko wtedy, gdy spełnimy oba warunki, więc możemy uzyskać ten sam efekt z operatorem `and`:

```
if 0 < x and x < 10:
    print('x jest jednocyfrową dodatnią liczbą.')
```

3.7. Łapanie wyjątków przy użyciu `try` i `except`

Wcześniej widzieliśmy fragment kodu, w którym używaliśmy funkcji `input()` i `int()` do odczytania i przetworzenia liczby całkowitej wprowadzonej przez użytkownika. Widzieliśmy też, jak takie podejście może być zdradzieckie:

```
>>> prompt = "Jaka jest prędkość lotu jaskółki bez obciążenia?\n"
>>> speed = input(prompt)
Jaka jest prędkość lotu jaskółki bez obciążenia?
Jakiej jaskółki? Afrykańskiej czy europejskiej?
>>> int(speed)
ValueError: invalid literal for int() with base 10:
>>>
```

Gdy wykonamy te polecenia w interpreterze Pythona, otrzymujemy od niego prośbę o wprowadzenie danych, interpreter pomyśli "ups!" i przejdzie do naszej następnej instrukcji.

Jeżeli jednak umieścisz ten kod w skrypcie Pythona i wystąpi błąd, to Twój skrypt natychmiast się zatrzyma w miejscu pojawienia się błędu, wyświetlając przy tym informacje z mechanizmu *traceback*, pozwalającego zobaczyć kolejne wywołania funkcji, które ostatecznie doprowadziły do wystąpienia błędu. Po wystąpieniu błędu skrypt nie wykona kolejnych instrukcji.

Oto przykładowy program do konwersji temperatury Fahrenheita na temperaturę Celsjusza:

```
inp = input('Podaj temperaturę w skali Fahrenheita: ')
fahr = float(inp)
cel = (fahr - 32.0) * 5.0 / 9.0
print(cel)

# Kod źródłowy: https://py4e.pl/code3/fahren.py
```

Jeśli uruchomimy ten kod i podamy mu błędne dane wejściowe, to program po prostu zakończy się niepowodzeniem z niemiłym komunikatem o błędzie:

```
Podaj temperaturę w skali Fahrenheita: 72
22.22222222222222
```

```
Podaj temperaturę w skali Fahrenheita: fred
Traceback (most recent call last):
  File "fahren.py", line 2, in <module>
    fahr = float(inp)
ValueError: could not convert string to float: 'fred'
```

W Pythonie jest wbudowana warunkowa struktura wykonania poleceń, która obsługuje tego typu oczekiwane i nieoczekiwane błędy, zwana "try / except". Idea `try` i `except` polega na tym, że wiesz, iż pewna sekwencja instrukcji może sprawić problem i chcesz dodać kilka poleceń do wykonania w przypadku wystąpienia błędu. Te dodatkowe polecenia (blok "except") są ignorowane, gdy nie ma błędu.

Możesz myśleć o `try` i `except` w Pythonie jako o "polisie ubezpieczeniowej" na sekwencję poleceń.

Możemy przepisać nasz konwerter temperatury w następujący sposób:

```
inp = input('Podaj temperaturę w skali Fahrenheita: ')
try:
    fahr = float(inp)
    cel = (fahr - 32.0) * 5.0 / 9.0
    print(cel)
except:
    print('Musisz wprowadzić liczbę')

# Kod źródłowy: https://py4e.pl/code3/fahren2.py
```

Python zaczyna od wykonania sekwencji poleceń w bloku `try`. Jeśli wszystko pójdzie dobrze, pomija blok `except` i przechodzi dalej. Jeśli wystąpi wyjątek w bloku `try`, Python wyskakuje z bloku `try` i wykonuje sekwencję instrukcji w bloku `except`.

```
Podaj temperaturę w skali Fahrenheita: 72
22.22222222222222
```

```
Podaj temperaturę w skali Fahrenheita: fred
Musisz wprowadzić liczbę
```

Obsługa wyjątku z instrukcją `try` nazywana jest *łapaniem* wyjątku. W tym przykładzie klauzula `except` wypisuje komunikat o błędzie. Ogólnie rzecz biorąc, złapanie wyjątku daje Ci szansę naprawienia problemu, spróbowania ponownie albo przynajmniej zgrabnego zakończenia programu.

3.8. Minimalna ewaluacja wyrażeń logicznych

Gdy Python przetwarza wyrażenie logiczne takie jak `x >= 2 and (x/y)> 2`, ewaluuje je od lewej do prawej strony. Ze względu na definicję `and`, jeśli `x` jest mniejsze niż 2, wyrażenie `x >= 2` zostanie ewaluowane do `False`, a więc całe wyrażenie zostanie ewaluowane do `False`, niezależnie od tego czy `(x/y) > 2` zostanie ewaluowane do `True` czy `False`.

Gdy Python wykryje, że nic już nie zyska dzięki ewaluacji pozostałej część wyrażenia logicznego, przerywa jego ewaluację i nie wykonuje dalszych obliczeń. Zatrzymanie oceny wyrażenia logicznego, ponieważ ogólna wartość jest już znana, nazywa się *minimalną ewaluacją*.

Konsekwencją specyfiki działania minimalnej ewaluacji jest sprytna technika zwana *wzorcem strażnika*. Przeanalizujmy następującą sekwencję kodu w interpreterze Pythona:

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

Trzecie obliczenie nie powiodło się, ponieważ Python ewaluował `(x/y)` i `y` było zerem, co spowodowało błąd wykonania. Ale pierwszy i drugi przykład *nie* zakończyły się błędami, ponieważ pierwsza część tych wyrażeń `x >= 2` została ewaluowana do `False`, więc warunek `(x/y)` nigdy nie został sprawdzony właśnie z powodu reguły minimalnej ewaluacji, przez co nie było tutaj żadnego błędu.

Możemy w następujący sposób skonstruować takie wyrażenie logiczne, aby strategicznie umieścić *strażnika* tuż przed ewaluacją potencjalnie powodującą błąd:

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

W pierwszym wyrażeniu logicznym, `x >= 2` wynosi `False`, więc ewaluacja kończy się na `and`. W drugim wyrażeniu logicznym, `x >= 2` wynosi `True`, ale `y != 0` wynosi `False`, więc nigdy nie dochodzimy do próby ewaluacji `(x/y)`.

W trzecim wyrażeniu logicznym, `y != 0` jest *po* obliczeniu `(x/y)`, więc wyrażenie kończy się błędem.

W drugim wyrażeniu mówimy, że `y != 0` działa jak *strażnik*, tak by zapewnić, że wykonamy `(x/y)` tylko wtedy, gdy `y` nie jest zerem.

3.9. Debugowanie

Mechanizm *traceback* w Pythonie pojawia się w przypadku wystąpienia błędu; zawiera on wiele informacji, ale czasem może być przytłaczający. Najbardziej użyteczne są zazwyczaj informacje o tym:

- jaki to był błąd,
- gdzie ten błąd wystąpił.

Błędy składniowe są zazwyczaj łatwe do znalezienia, ale jest kilka podstępnych przypadków. Błędy związane z tzw. białymi znakami[2] mogą być dla nas trudne do wychwycenia, ponieważ np. spacje i tabulacje są mało widoczne i jesteśmy przyzwyczajeni do ich ignorowania.

[2]Należą do nich spacje, znaki tabulacji, znaki końca linii oraz dowolne inne znaki niemające kształtu na ekranie.

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

W powyższym przykładzie problem polega na tym, że druga linia jest wcięta przez jedną spację. Ale komunikat o błędzie wskazuje na `y`, co jest mylące. Ogólnie rzecz biorąc, komunikaty o błędach wskazują, gdzie problem został wykryty, ale rzeczywisty błąd może być w kodzie gdzieś wcześniej, czasami w poprzedniej linii.

Ogólnie komunikaty o błędach informują o tym, gdzie wykryto problem, ale często nie tam, gdzie został on popełniony.

3.10. Słowniczek

ciało instrukcji Sekwencja instrukcji zawarta w złożonej instrukcji.
gałąź Jedna z alternatywnych sekwencji instrukcji występujących w instrukcji warunkowej.
instrukcja warunkowa Instrukcja, która kontroluje przepływ wykonywania programu w zależności od pewnego warunku.
instrukcja złożona Instrukcja składająca się z nagłówka i ciała. Nagłówek kończy się dwukropkiem (`:`). Ciało jest wcięte w stosunku do nagłówka.
minimalna ewaluacja Kiedy Python jest w trakcie ewaluacji złożonego wyrażenia logicznego i przerywa dalszą ewaluację, ponieważ Python zna końcową wartość wyrażenia bez potrzeby ewaluacji reszty wyrażenia.
operator logiczny Jeden z operatorów, który łączy wyrażenia logiczne: `and`, `or` lub `not`.
operator porównania Jeden z następujących operatorów, który porównuje swoje operandy: `==`, `!=`, `>`, `<`, `>=` i `<=`.
traceback Lista funkcji, które są wykonywane; pojawia się, gdy wystąpi wyjątek.
warunek Wyrażenie logiczne w instrukcji warunkowej określające która gałąź jest wykonywana.
warunki powiązane Wyrażenie warunkowe z serią alternatywnych gałęzi.
warunki zagnieżdżone Instrukcja warunkowa, która pojawia się w jednej z gałęzi innej instrukcji warunkowej.
wyrażenie logiczne Wyrażenie o wartości `True` lub `False`.
wzorzec strażnika Miejsce, w którym konstruujemy wyrażenie logiczne z dodatkowymi porównaniami, tak aby skorzystać z działania minimalnej ewaluacji.

3.11. Ćwiczenia

Ćwiczenie 1. Przepisz ponownie swój program obliczający wynagrodzenie, tak aby dać pracownikowi 1,5 raza większą stawkę godzinową za czas przepracowany powyżej 40 godzin.

```
Podaj liczbę godzin: 45
Podaj stawkę godzinową: 10
Wynagrodzenie: 475.0
```

Ćwiczenie 2. Przepisz ponownie swój program płacowy, używając `try` i `except`, tak aby elegancko obsługiwał on nienumeryczne dane wejściowe, wyświetlając w takim przypadku wiadomość i kończąc swoje działanie. Poniżej znajdują się wyniki dwóch uruchomień programu:

```
Podaj liczbę godzin: 20
Podaj stawkę godzinową: dziewięć
Błąd, podaj wartość numeryczną
```

```
Podaj liczbę godzin: czterdzieści
Błąd, podaj wartość numeryczną
```

Ćwiczenie 3. Napisz program, który poprosi użytkownika o wartość pomiędzy 0.0 a 1.0. Jeśli wartość jest poza zakresem, wypisz komunikat o błędzie. Jeśli wartość jest między 0.0 a 1.0, wypisz ocenę, korzystając z poniższej tabeli:

```
Wartość   Ocena
>= 0.9     5,0
>= 0.8     4,5
>= 0.7     4,0
>= 0.6     3,5
>= 0.5     3,0
 < 0.5     2,0
```

```
Podaj wartość: 0.95
5,0
```

```
Podaj wartość: doskonała
Niepoprawna wartość
```

```
Podaj wartość: 10.0
Niepoprawna wartość
```

```
Podaj wartość: 0.75
4,0
```

```
Podaj wartość: 0.5
3,0
```

```
Podaj wartość: 0.46
2,0
```

Uruchom program wielokrotnie, jak pokazano powyżej, tak aby przetestować różne wartości.