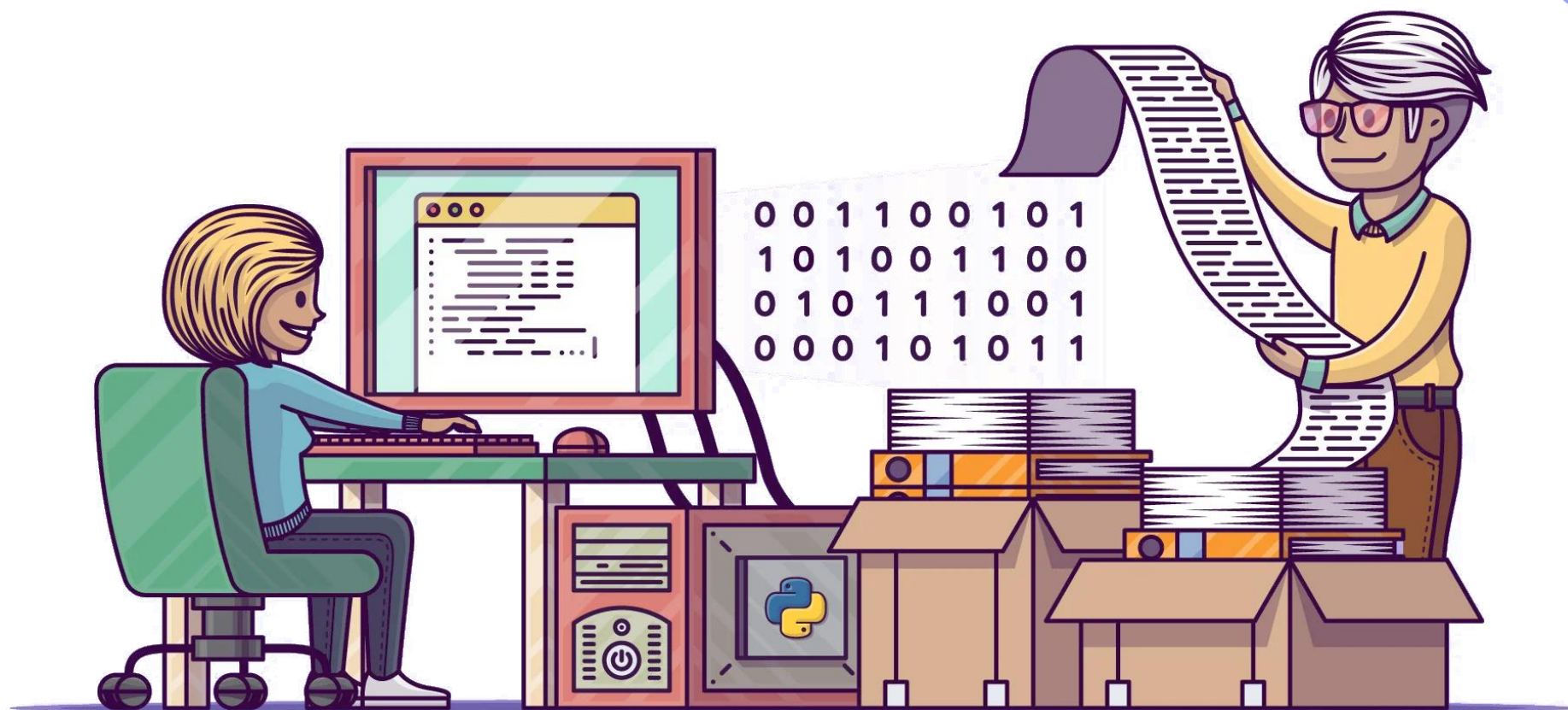


# Pliki



Real Python

# Wczytywanie zawartości pliku



Otwarcie pliku w trybie do odczytu

```
f = open("demo.txt", "r")
```

# Wczytywanie zawartości pliku



Otwarcie pliku w trybie do odczytu

```
f = open("demo.txt", "r")
```

Wczytanie zawartości pliku

```
content = f.read()
```

# Wczytywanie zawartości pliku



Otwarcie pliku w trybie do odczytu

```
f = open("demo.txt", "r")
```

Wczytanie zawartości pliku

```
content = f.read()
```

Zamknięcie pliku

```
f.close()
```

# Zapisywanie do pliku



Otwarcie pliku w trybie do zapisu

```
f = open('demo.txt', 'w')
```

# Zapisywanie do pliku



Otwarcie pliku w trybie do zapisu

```
f = open('demo.txt', 'w')
```

Zapisanie do pliku

```
f.write("Ała ma kota")
```

# Zapisywanie do pliku



Otwarcie pliku w trybie do zapisu

```
f = open('demo.txt', 'w')
```

Zapisanie do pliku

```
f.write("Ała ma kota")
```

Zamknięcie pliku

```
f.close()
```

# Tryby otwierania pliku

(drugi parametr funkcji open)



Odczyt:

**r – otwarcie pliku w trybie do odczytu (tworzy plik jeżeli taki plik nie istnieje)**

Zapis:

a – otwarcie pliku w trybie do dopisywania (tworzy plik jeżeli taki plik nie istnieje)

w – otwarcie pliku w trybie do nadpisywania (tworzy plik jeżeli taki plik nie istnieje)

x – stworzenie pliku (rzuca błąd, jeżeli plik już istnieje)



# Tryby otwierania pliku

(drugi parametr funkcji open)



**t – tryb tekstowy**

b – tryb binarny

# Menadžer kontekstu



Real Python

# Wczytywanie zawartości pliku

(menadżer kontekstu)



```
with open('demo.txt', 'r') as f:  
    content = f.read()
```

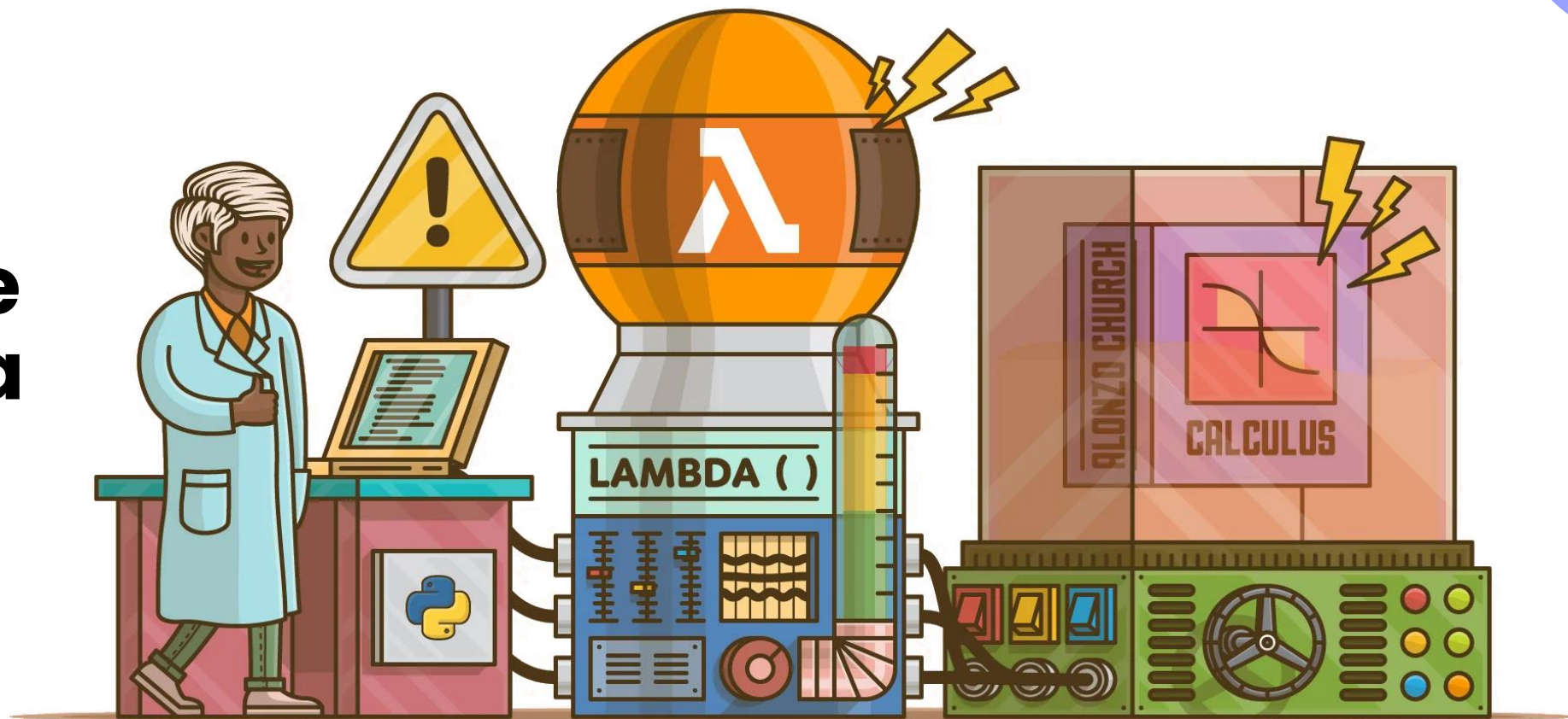
# Zapisywanie do pliku

(menadżer kontekstu)



```
with open('demo.txt', 'w') as f:  
    content = f.write("Ała ma kota")
```

# Funkcje lambda



Real Python

# Funkcje lambda



Zwykła funkcja

```
def square(x):  
    return x * x
```

Funkcja lambda

```
square = lambda x: x * x
```

# Funkcje lambda



Zwykła funkcja

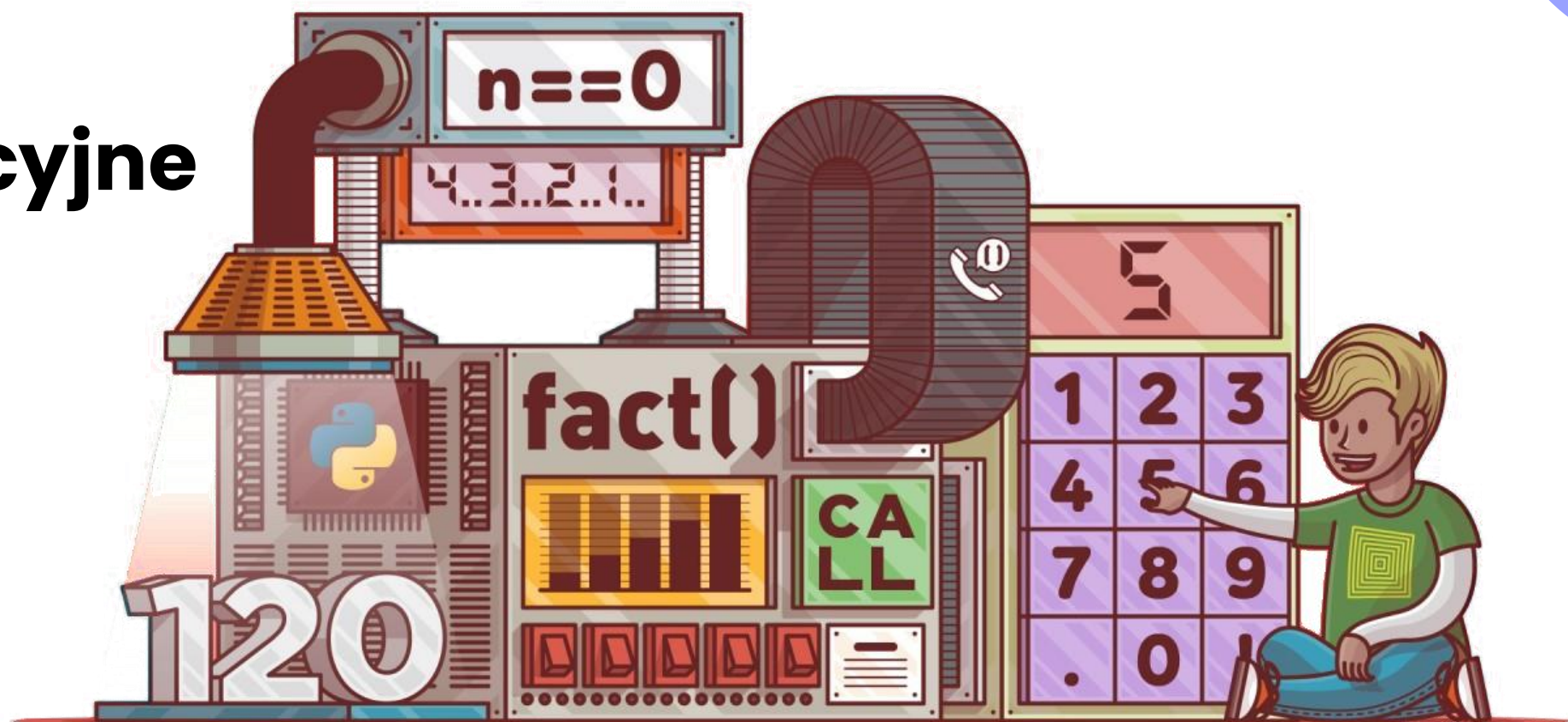
```
def sum_(x, y):  
    return x + y
```

Funkcja lambda

```
sum_ = lambda x, y: x + y
```



# Funkcje rekurencyjne



Real Python



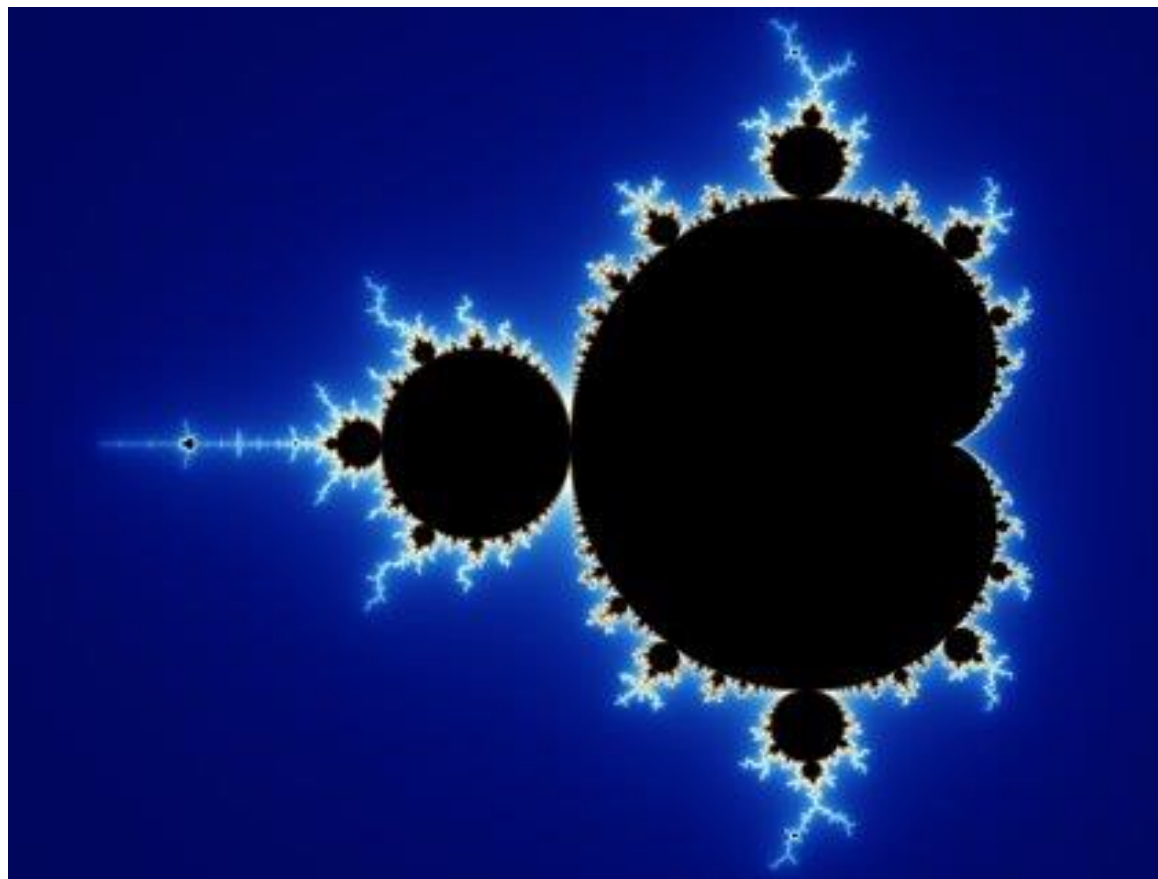
# Rekurencja



# Rekurencja



# Rekurencja





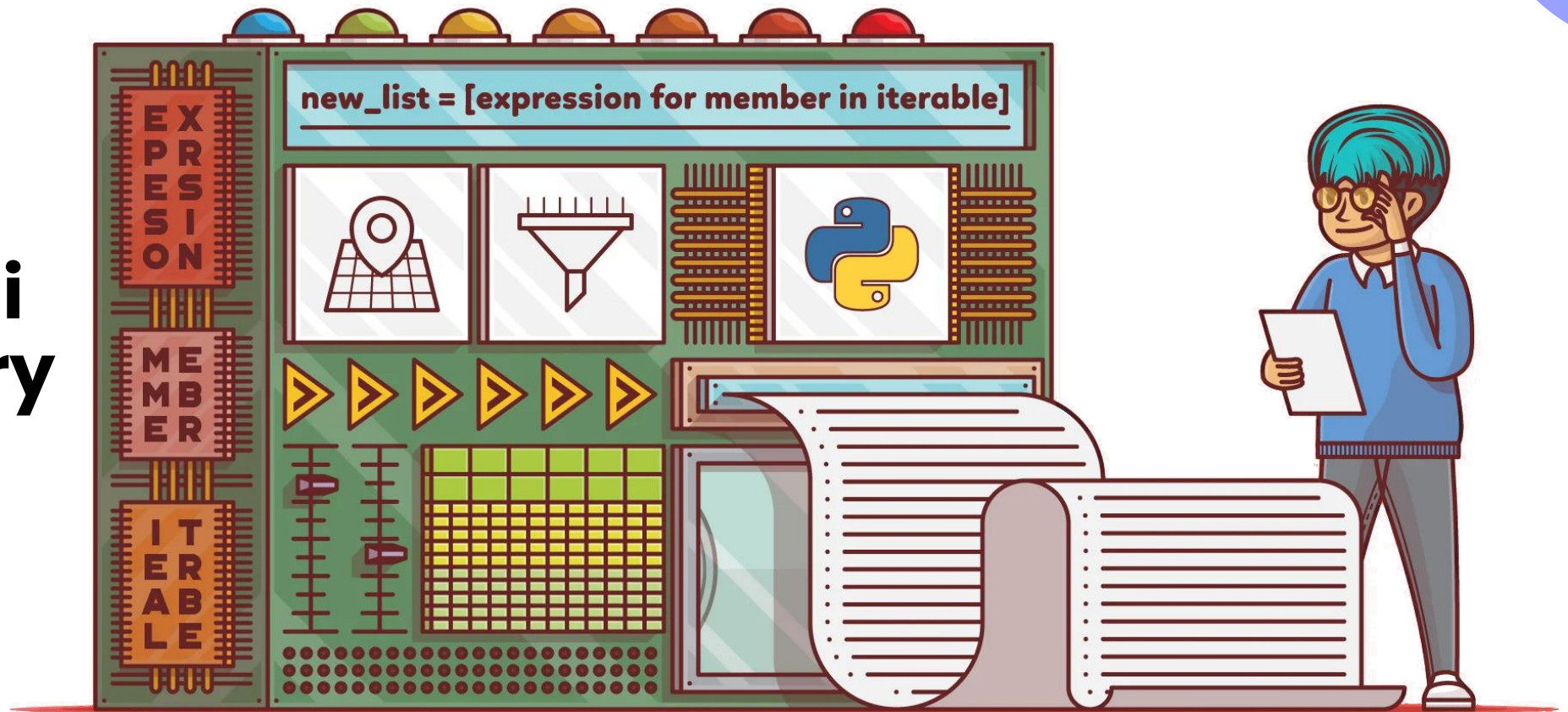
An illustration of a man in a dark suit, light blue shirt, and pink tie standing behind a service counter. He is looking down at a small green card in his hand. In front of him, a queue of four people is waiting, their backs to the viewer. The background is a plain tan wall with vertical lines. A sign hangs from the ceiling above the man.

`x == 1`

```
def count(x):  
    if x == 1:  
        return 1  
    else:  
        return x + count(x-1)
```



# Listy składane i generatory



Real Python

# Listy składane

(list comprehension)



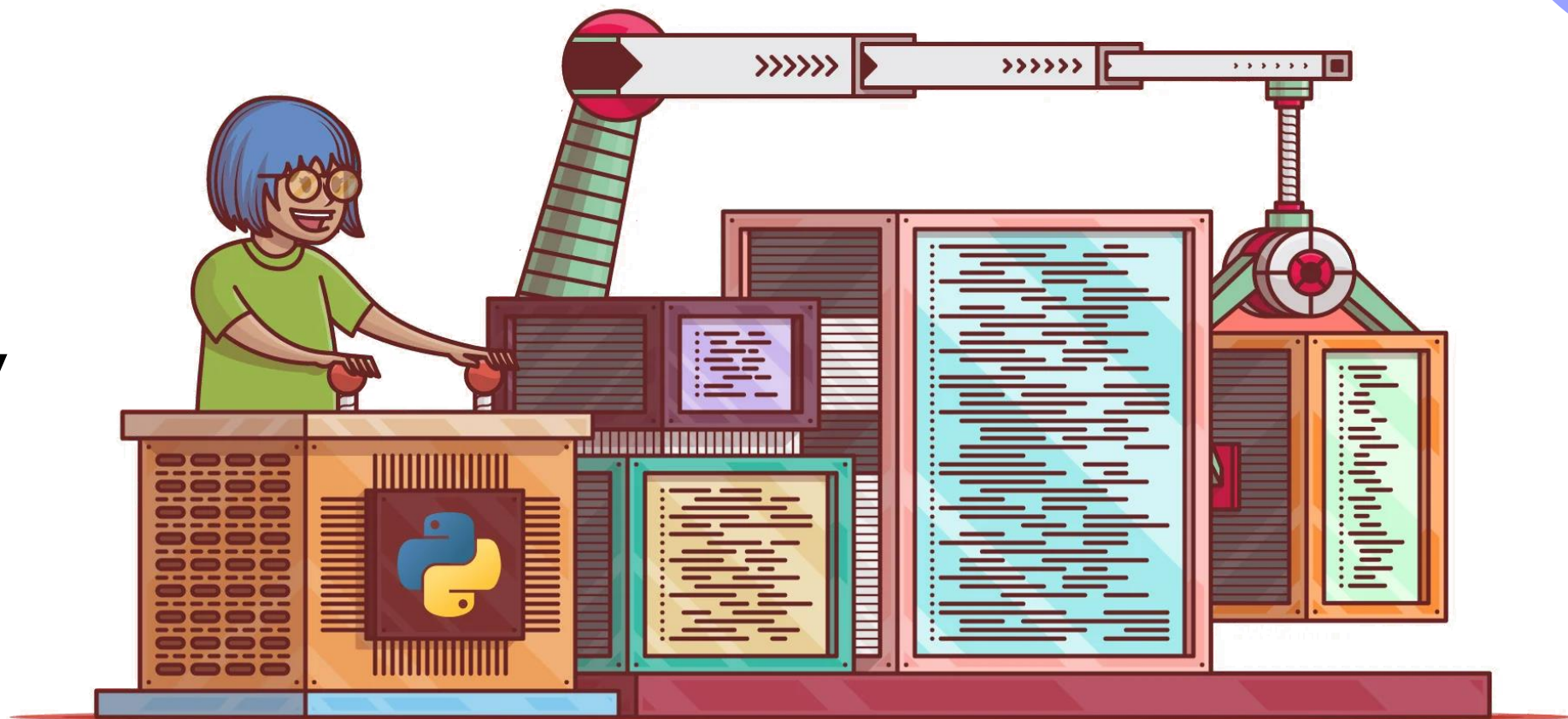
Tworzenie listy z wykorzystaniem zwykłej pętli

```
even_numbers = []  
for item in range(10):  
    if item % 2 == 0:  
        even_numbers.append(item)
```

Tworzenie listy za pomocą składania (comprehension)

```
even_numbers = [item for item in range(10) if item % 2 == 0]
```

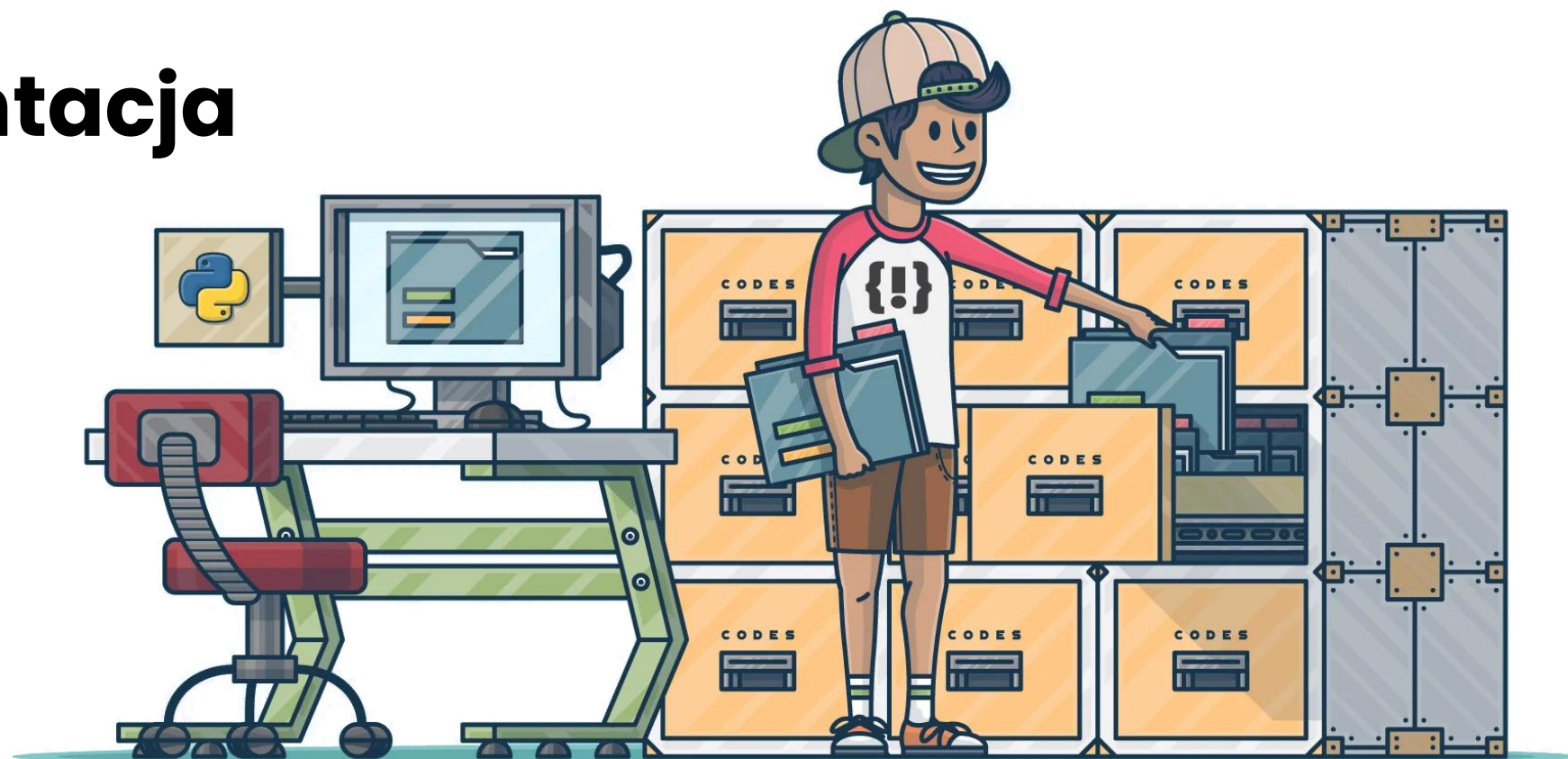
# Moduły



Real Python



# Dokumentacja



Real Python

# Wyjątki





# Obsługa wyjątków

# Obsługa wyjątków



Obsługa (przechwytywanie) wyjątków oznacza reagowanie na wystąpienie błędu. W Pythonie istnieją dwa podstawowe paradygmaty (sposoby) postępowania z błędami, spopularyzowane za pomocą akronimów:

1. **LBYL** – **L**ook **B**efore **Y**ou **L**eap
2. **EAFP** – **E**asier to **A**sk **F**orgiveness Than **P**ermission

LBYL oznacza przeciwdziałanie wystąpieniu błędu (prewencje), EAFP oznacza reagowanie na jego wystąpienie (leczenie). Odpowiedź na pytanie czy w Pythonie lepiej jest przeciwdziałać, czy leczyć nie jest tak oczywista jak to jest w medycynie. W zależności od zastosowania jeden ze sposobów przeważnie okazuje się bardziej odpowiedni od drugiego.

Wyjątki są realizacją w Python-ie tej drugiej metody postępowania - **EAFP**.



# Obsługa wyjątków

Wróćmy do jednego z naszych pierwszych zadań.

```
from math import pi

def circle_area(r):
    """Return area of the circle for the given radius."""
    return pi * r**2

raw_r = input("Proszę, podaj mi promień koła: ")
r = float(raw_r)
area = circle_area(r)
print(f"Pole koła o promieniu {r} wynosi {round(area, 2)}")
```

Co możemy zrobić, żeby ustrzec się przed niepoprawnymi wartościami promienia podawanymi przez użytkownika ?

# Obsługa wyjątków



Pierwszym pomysłem może być sprawdzanie jaką wartość przekazał użytkownik do programu (LBYL).

```
from math import pi

def circle_area(r):
    """Return area of the circle for the given radius."""
    return pi * r**2

raw_r = input("Proszę, podaj mi promień koła: ")

if not raw_r.isnumeric():
    print("Nie podałeś wartości liczbowej.")
else:
    r = float(raw_r)
    area = circle_area(r)
    print(f"Pole koła o promieniu {r} wynosi {round(area, 2)}")
```

Używamy instrukcji warunkowej.

W zależności od wyniku wypisujemy komunikat o niepoprawnych danych wejściowych lub wykonujemy dalej nasz program. W takim schemacie nasz warunek przyjmuje na siebie zadanie pytania o pozwolenie na wykonanie kodu (przeciwdziałamy wystąpieniu błędu).



# Obsługa wyjątków

Zamiast pytać o pozwolenie możemy niczym się nie przejmując wykonać nasz program i odpowiednio zareagować jeżeli coś pójdzie nie tak (**EAFP**). Właśnie do tego celu w Python-ie wprowadzono **wyjątki**.

```
from math import pi

def circle_area(r):
    """Return area of the circle for the given radius."""
    return pi * r**2

raw_r = input("Proszę, podaj mi promień koła: ")

try:
    r = float(raw_r)
    area = circle_area(r)
    print(f"Pole koła o promieniu {r} wynosi {round(area, 2)}")
except:
    print("Nie podałeś wartości liczbowej.")
```

Używamy bloku **try-except**.

Wewnątrz instrukcji **try** umieszczamy kod, którego wywołanie może spowodować błąd (wyjątek). Wewnątrz instrukcji **except** umieszczamy kod, który ma zostać wykonany jeżeli podczas wykonywania kodu z instrukcji **try** wystąpi błąd (wyjątek).

W tym schemacie nie pytamy się o pozwolenie. Wykonujemy nasz kod, a jeżeli wystąpi błąd (wyjątek) odpowiednio ten błąd (wyjątek) obsługujemy (wewnątrz instrukcji **except**).



# Pokemon i Yoda

```
from math import pi

def circle_area(r):
    """Return area of the circle for the given radius."""
    return pi * r**2

raw_r = input("Proszę, podaj mi promień koła: ")

try:
    r = float(raw_r)
    area = circle_area(r)
    print(f"Pole koła o promieniu {r} wynosi {round(area, 2)}")

except:
    print("Nie podałeś wartości liczbowej.")
```

Pokemon exception handling - "Pokemon – gotta catch 'em all"

Yoda exception handling - "Do or do not. There is not try." (There is no partial success)

Diaper Pattern - "catches all the shit"



# Typy błędów



W Pythonie mamy dwa główne typy błędów:

1. Błędy składni (Syntax Errors)
2. Wyjątki (Exceptions)

**Błędy składni** są to błędy wywoływane przy próbie uruchomienia programu, którego kod posiada nieprawidłową składnię. Przykładami takich błędów mogą być: nieprawidłowa głębokość wcięcia w kodzie, brak dwukropka, brak nawiasu zamykającego lub jednego z dwóch apostrofów napisu. Tego typu błędy są sygnalizowane przed właściwym uruchomieniem kodu i uniemożliwiają jego wykonanie. Kod programu ma składnię niezgodną ze składnią języka Python i program w ogóle nie może być uruchomiony za pomocą interpretera Python. Jedyny sposób naprawienia tego typu błędów to znalezienie miejsca w kodzie z nieprawidłową składnią i poprawienie go. Często pomocny jest tu traceback wyświetlany przez interpreter Pythona.

Drugi typ błędów to błędy wywoływane celowo przez twórców bibliotek w celu zasygnalizowania wystąpienia niepożądanego scenariusza. Właśnie takie błędy nazywamy **wyjątkami**. W Pythonie istnieje cała hierarchia wyjątków, której nie będziemy szczegółowo omawiać. Do najpopularniejszych (z wbudowanych) typów wyjątków należą m.in.:

- TypeError
- ValueError
- ZeroDivisionError
- IndexError
- KeyError

Python rzuca odpowiedni wyjątek w przypadku wystąpienia niepożądanego scenariusza.

# Obsługa wyjątków



Pusta instrukcja `except`, której użyliśmy w poprzednim przykładzie jest zbyt ogólna. Znacznie lepiej jest wskazać dokładny typ wyjątku jaki chcemy obsłużyć. W naszym przykładzie Python, w wyniku błędu rzutowania na float podniesie wyjątek typu `ValueError`. Obsłużmy dokładnie ten typ wyjątku (a nie tak jak wcześniej - wszystkie wyjątki, które mogłyby wystąpić)

```
from math import pi

def circle_area(r):
    """Return area of the circle for the given radius."""
    return pi * r**2

raw_r = input("Proszę, podaj mi promień koła: ")

try:
    r = float(raw_r)
    area = circle_area(r)
    print(f"Pole koła o promieniu {r} wynosi {round(area, 2)}")
except ValueError:
    print("Nie podałeś wartości liczbowej.")
```

# Obsługa wyjątków



W naszym kodzie, powinniśmy zadbać o właściwe obsłużenie **wszystkich** miejsc, w których mogą wystąpić wyjątki. Jeżeli coś poszło nie tak, użytkownik naszego programu powinien dostać czytelny komunikat co się stało.

# Obsługa wyjątków



Wyjątki możemy używać również w inny sposób. Jako twórcy kodu (czyli często bibliotek) możemy je samodzielnie wywoływać w określonych scenariuszach. Wywoływanie wyjątku nazywamy rzucaniem wyjątku.

# Rzucanie wyjątków



# Rzucanie wyjątków

Rzucenie (podnoszenie) wyjątku oznacza celowe wywołanie wyjątku w naszym kodzie w celu zasygnalizowania wystąpienia niepożądanego scenariusza.

W Pythonie do rzucenia wyjątku służy instrukcja `raise`, po której następuje nazwa rzucanego wyjątku, np.:

```
from math import pi

def circle_area(r):
    """Return area of the circle for the given radius."""
    if r == 0:
        raise ValueError
    return pi * r**2
```



# Rzucanie wyjątków

Mówiąc o obsłudze wyjątków przyjęliśmy perspektywę użytkownika 'biblioteki'. Używamy wbudowanego kodu, który rzuca wyjątek, kiedy wystąpi niepożądany scenariusz. Naszym zadaniem jest obsługa takiego wyjątku. Drugą stroną tej samej monety jest rzucanie wyjątków.

Na to samo zagadnienie możemy patrzeć z perspektywy twórcy 'biblioteki'. Pisząc kod sami stajemy się takimi twórcami. Możliwe, że w przypadku wystąpienia niepożądanego scenariusza, my jako twórcy biblioteki sami chcielibyśmy wywołać (mówimy podnieść/rzucić) wyjątek. W takiej sytuacji odpowiedzialność za obsłużenie rzuconego przez nas wyjątku spoczywa na użytkownikach naszego kodu ('biblioteki').

Często jesteśmy twórcami oraz użytkownikami kodu jednocześnie. Na przykład piszemy funkcję (jesteśmy twórcami tej funkcji), a potem stworzoną funkcję wykorzystujemy w różnych miejscach w naszym kodzie (jesteśmy użytkownikami tej funkcji).



# Rzucanie wyjątków

Po nazwie wyjątku, wewnątrz nawiasów można dodać opcjonalnych komunikat, który zostanie wyświetlony w przypadku podniesienia wyjątku (realizacji scenariusza, który wywoła wyjątek).

```
from math import pi

def circle_area(r):
    """Return area of the circle for the given radius."""

    if r == 0:
        raise ValueError("Circle with the diameter 0 is not allowed")

    return pi * r**2
```





# Rzucanie wyjątków

Wyjątki rzuca się z myślą o kodzie klienckim, którego zadaniem będzie obsłużenie tego wyjątku.

```
from math import pi

def circle_area(r):
    """Return area of the circle for the given radius."""
    if r == 0:
        raise ValueError("Circle with the diameter 0 is not allowed")
    return pi * r**2
```

Kod kliencki

```
raw_r = input("Proszę, podaj mi promień koła: ")

try:
    r = float(raw_r)
    area = circle_area(r)
    print(f"Pole koła o promieniu {r} wynosi {round(area, 2)}")

except ValueError:
    print("Nie istnieje koło o promieniu 0.")
```

# Rzucanie wyjątków



Wyjątki rzuca się z myślą o kodzie klienckim, którego zadaniem będzie obsłużenie tego wyjątku.

```
from math import pi

def circle_area(r):
    """Return area of the circle for the given radius."""
    if r == 0:
        raise ValueError("Circle with the diameter 0 is not allowed")
    return pi * r**2
```

Rzucenie wyjątku

```
raw_r = input("Proszę, podaj mi promień koła: ")

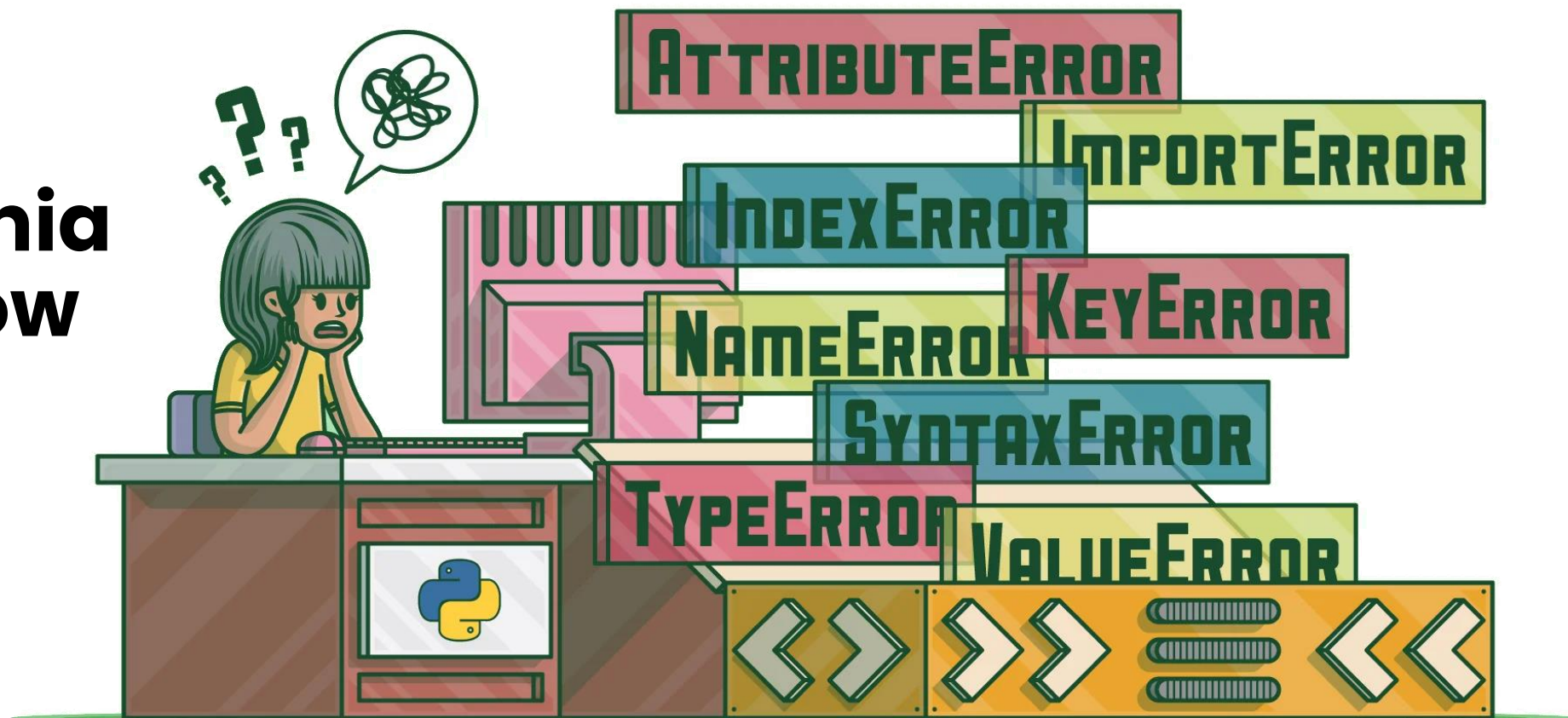
try:
    r = float(raw_r)
    area = circle_area(r)
    print(f"Pole koła o promieniu {r} wynosi {round(area, 2)}")

except ValueError:
    print("Nie istnieje koło o promieniu 0.")
```

Kod kliencki

Obsługa wyjątku

# Hierarchia wyjątków



Real Python

# Dziękujemy!

