Datentypen

str	"Hallo Welt", "", "Ferien!"
int	0, 1, 2, 3, 42, -5
float	0.0, 3.14
bool	True, False
	Vergleiche durch a == b, !=, <, >, <=, >=
tuple	(1, 2), (255,255,255), ()
list	[], [1, 2, 3], ["a", "b", "c"]
set	{}, {1,2,3} (ohne Doppelungen)
dictionary	{key: value,...} {name:"Gerda", vorname:"Taro"}
optional	Wert oder None (kein Wert)

Liste

```
emptyList = []
names = ["Gauß", "Euler", "Fermat"]
primes = [2, 3, 5, 7, 11]

# Zugriff auf Listenelemente/Werte
firstPrime = primes[0]
secondPrime = primes[1]
lastPrime = primes[-1]
lastPrime = primes[len(primes)-1]

# Slicing
primes[von:bis], primes[von:bis:schritt], primes[2:4] # [5, 7]
```

Methoden der Klasse List

```
# fügt ein Element vom Typ Object in Liste ein
append(element: Object): None

# fügt ein Element an der Stelle index ein
insert(index:int, element: Object): None

# Löscht einen Wert
remove(element: Object):None

# Löscht das letzte Element
pop(): Object # get and remove last element

# Löscht ein Element an einem Index und gibt es zurück
pop(index: int): Object

# sortiert die Liste
sort(): None

# gibt Länge zurück
__len__(): int
```

Bedingungen

Haben den Datentyp bool. Werden häufig durch Vergleiche ausgerechnet und für if-Verzweigung / while-Schleife verwendet.

```
x == 42    x < 42    x > 42    42 in [2, 3, 5, 7, 11]
x != 42    x <= 42   x >= 42
```

2 Bedingungen können mit den logischen Operatoren **and** und **or** verknüpft werden.

if-Verzweigung

Für Code, der nur unter bestimmten Bedingungen ausgeführt werden soll.

Allgemein

```
if bedingung:
        # ...
elif bedingung:
        # ...
else:
        # ...
```

```
if age >= 18:
        print("Du darfst wählen.")
```

```
if age < 4:
        ticket_price = 0
elif age < 18:
        ticket_price = 10
else:
        ticket_price = 15
```

while-Schleife

Für Wiederholungen, wenn die Anzahl der Durchläufe von einer Bedingung abhängig ist.

Allgemein

```
while bedingung:
    # ...
    # vorzeitiges Verlassen mit break oder continue möglich
```

```
while True:
    print("Ich bin toll!")
```

```
while entfernungZumZiel > 0:
    schritt()
```

for-Schleife

Für Wiederholungen, wenn

- die Anzahl der Durchläufe bekannt ist
- über eine Liste iteriert wird

Allgemein

```
for variable in liste:
        # ...
```

Wird nicht über eine Liste iteriert, so kann die range Funktion verwendet werden. Diese erzeugt eine Liste.

range(bis) / range(von, bis) / range(von, bis , schritt)

```
range(6)        # [0, 1, 2, 3, 4, 5]
range(3, 6)     # [3, 4, 5]
range(0, 6, 2)  # [0, 2, 4]
```

```
for index in range(6):
        print(index)
```

```
for primzahl in [2, 3, 5, 7, 11]:
        print(primzahl)
```

Bereitgestellte Methoden / Funktionen

```
print(): None
input(text:str): str

str(Object):str
int(Object):int
bool(Object):bool
float(Object):float

len(liste): int
min(liste):int
max(liste):int

abs(n):int # absolute Wert
type(Object): str # gibt Typ
```

Eigene Methoden / Funktionen

Allgemein

```
def methoden_name(parameter1, parameter2, ...):
        # mache etwas
        return # gibt Ergebnis zurück
# Aufruf der Methode
methoden_name(argument1, argument2, ...)
```

```
def sageHallo():
        print("hallo")
sageHallo()
```

```
def sageHallo(name):
        print("hallo" + name  + "!")
sageHallo("Jürgen")
```

```
def summe(a, b):
        return a + b
ergebnis = summe(5,6)
```

Klassen

Allgemein

```
class Klassenname(Elternklasse):
    # Konstuktor
    def __init__(self, parameter1, parameter2, …):
        self. attributname = wert
        # Für private attribute
        self. __attributname = wert
    def methodenname(self, parameter1, parameter2, …):
        # Mache etwas

# Erzeugen eines Objekts der Klasse
einObjekt = Klassenname(argument1, argument2, …)
```

```
class Tier:
    def __init__(self, name, alter):
        self.name = name
        self.alter = alter
    def geburtstag_feiern(self):
        self.alter += 1
    def geraeusch_machen(self):
        # Diese Methode soll in Kindklassen überschrieben werden
        pass
    def __str__(self):
        return "Name: " + self.name + " Alter: " + str(self.alter)
```

Vererbung

```
# Kindklasse Hund
class Hund(Tier):
    def __init__(self, name, alter):
        super().__init__(name, alter)
    def geraeusch_machen(self):
        print("Wuff!")

# Kindklasse Katze
class Katze(Tier):
    def __init__(self, name, alter):
        super().__init__(name, alter)
    def geraeusch_machen(self):
        print("Miau!")

hund1 = Hund("Bello", 3)
katze1 = Katze("Luna", 2)
print(hund1)
hund1.geraeusch_machen()
hund1.geburtstag_feiern()

print(katze1)
katze1.geraeusch_machen()
katze1.geburtstag_feiern()
```

Kapselung

Inneren Daten und Implementierungsdetails einer Klasse werden nach außen verborgen.

Der Zugriff auf Attribute erfolgt in der Regel über Methoden (Getter/Setter), sodass man kontrollieren kann, wie Daten gelesen oder verändert werden

In Python sind laut Konvention alle Attribute public. Attribute, deren Name mit __ beginnt sind private und vor Zugriff von außen geschützt.

```
class Bankkonto:
    def __init__(self, inhaber, startguthaben=0):
        self.inhaber = inhaber
        # "privates" Attribut
        self.__guthaben = startguthaben

    def einzahlen(self, betrag):
        if betrag > 0:
            self.__guthaben += betrag
        else:
            print("Ungültiger Betrag!")

    def abheben(self, betrag):
        if betrag > 0 and betrag >= self.__guthaben:
            self.__guthaben -= betrag
        else:
            print("Abhebung nicht möglich!")

    def kontostand(self):
        return self.__guthaben

# Beispielverwendung
konto = Bankkonto("Alice", 100)
konto.einzahlen(50)
konto.abheben(30)
print("Aktueller Kontostand:", konto.kontostand())

# Direkter Zugriff scheitert:
# print(konto.__guthaben)   # AttributeError!
```

Polymorphie

Verschiedene Klassen bieten gleich Schnittstelle (gleich Methodennamen), aber jeweils eine eigene Umsetzung haben. Dadurch kann man Objekte unterschiedlicher Klassen gleich behandeln, obwohl sie sich im Verhalten unterscheiden.

Beispiel: Sowohl Hund als auch Katze haben eine Methode geraeusch_machen(). Ruft man diese Methode auf, bellt der Hund und die Katze miaut – also gleicher Aufruf, aber unterschiedliche Umsetzung.

Operatoren

Addition	a + b	Division	a / b
Subtraktion	a - b	Modulo	a % b
Multiplikation	a * b	Division ohne Rest	a // b

Zufall

```
import random
```

random()	Gibt eine Zufallszahl im Intervall [0.0, 1.0) zurück.
randint(a, b)	Gibt eine ganzzahlige Zufallszahl zwischen a und b zurück.
choice(seq)	Wählt zufälliges Element aus einer Sequenz (list, tuple, …).
choices(seq, k=n)	Gibt eine Liste von n Elementen zurück (mit Zurücklegen, d.h. Wiederholungen möglich).
gauss(mu, sigma)	Liefert eine Zufallszahl nach **Normalverteilung** mit Mittelwert mu und Standardabweichung sigma.
sample(seq, k)	Gibt k **verschiedene** zufällige Elemente aus einer Sequenz zurück (ohne Zurücklegen).
seed(x)	Setzt den Startwert des Zufallszahlengenerators (für Reproduzierbarkeit).

Freier Raum für Notizen ☺