合肥大學
HEFEI UNIVERSITY

Programming with Python

29. Funktionsargumente

Thomas Weise (汤卫思)
tweise@hfuu.edu.cn

Institute of Applied Optimization (IAO)
School of Artificial Intelligence and Big Data
Hefei University
Hefei, Anhui, China

应用优化研究所
人工智能与大数据学院
合肥大学
中国安徽省合肥市

Version: 2025-10-30

Programming with Python

Dies ist ein Kurs über das Programmieren mit der Programmiersprache Python an der Universität Hefei (合肥大学).

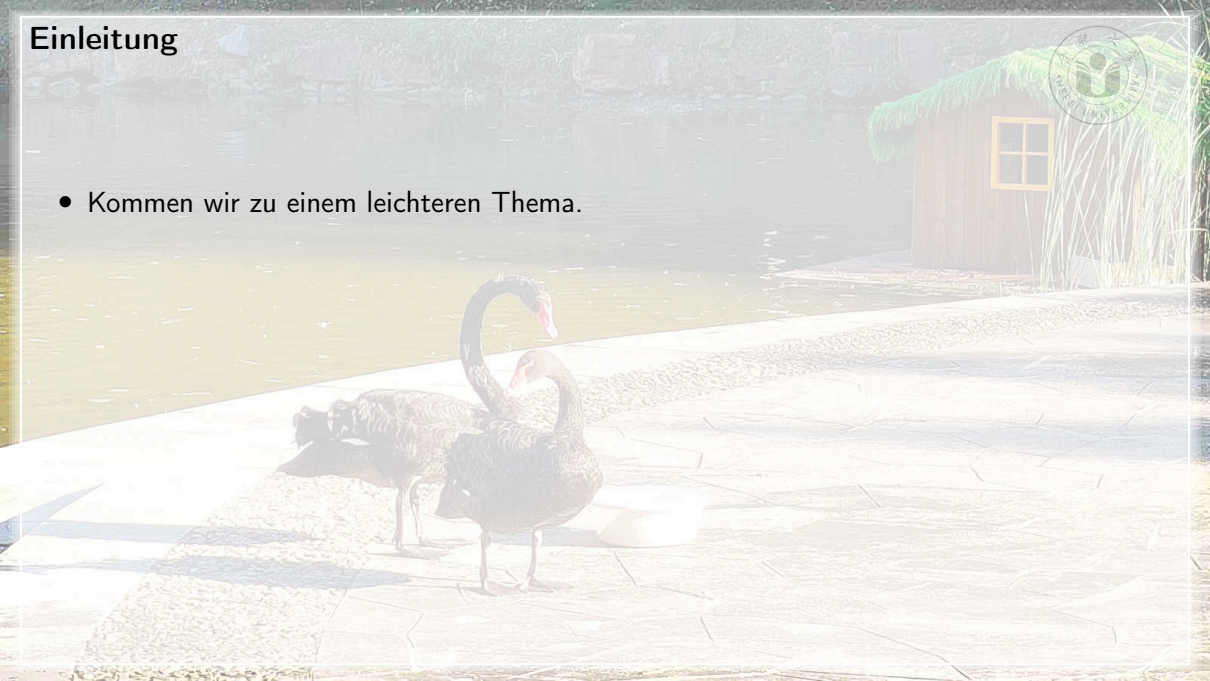
Die Webseite mit dem Lehrmaterial dieses Kurses ist https://thomasweise.github.io/programmingWithPython (siehe auch den QR-Kode unten rechts). Dort können Sie das Kursbuch (in Englisch) und diese Slides finden. Das Repository mit den Beispielprogrammen in Python finden Sie unter https://github.com/thomasWeise/programmingWithPythonCode.

Outline

Einleitung

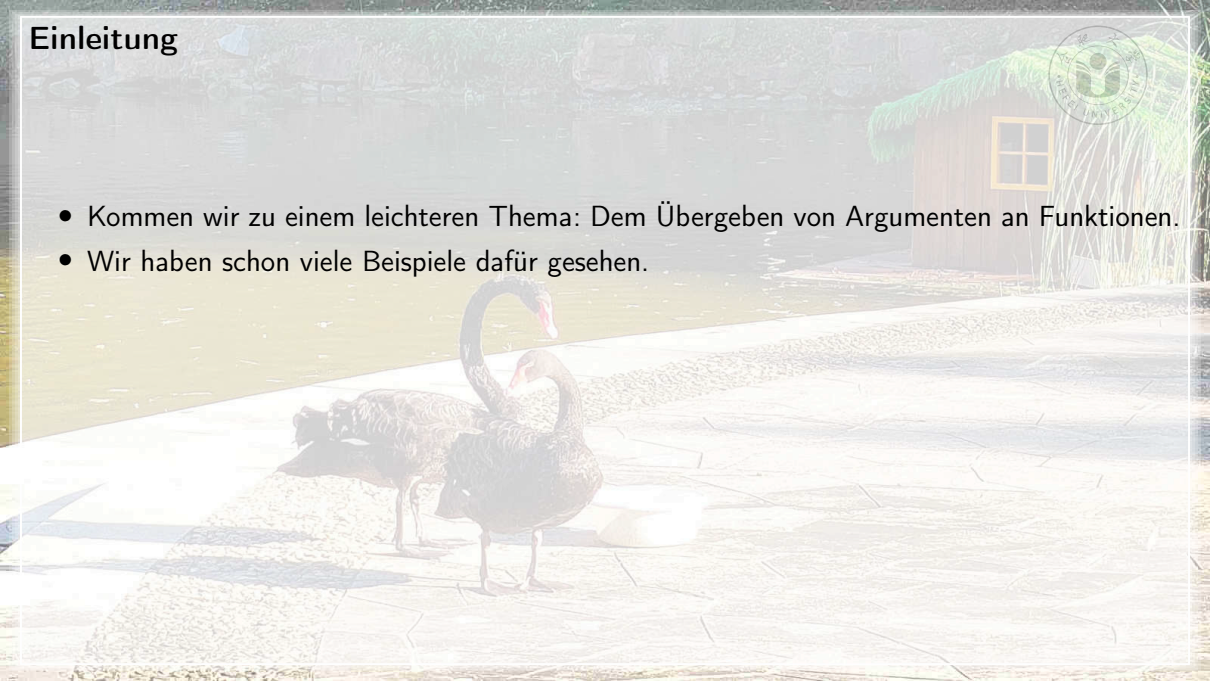
Einleitung

- Kommen wir zu einem leichteren Thema.



Einleitung

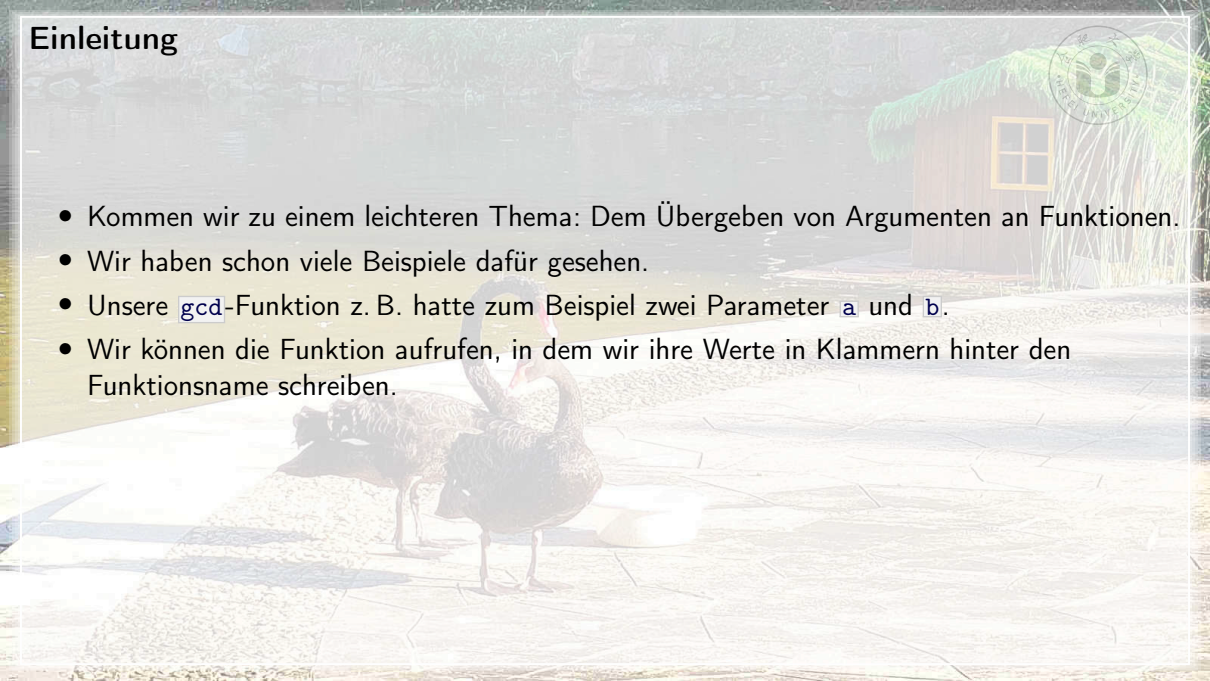
- Kommen wir zu einem leichteren Thema: Dem Übergeben von Argumenten an Funktionen.
- Wir haben schon viele Beispiele dafür gesehen.



Einleitung

- Kommen wir zu einem leichteren Thema: Dem Übergeben von Argumenten an Funktionen.
- Wir haben schon viele Beispiele dafür gesehen.
- Unsere `gcd`-Funktion z. B. hatte zum Beispiel zwei Parameter `a` und `b`.

Einleitung



- Kommen wir zu einem leichteren Thema: Dem Übergeben von Argumenten an Funktionen.
- Wir haben schon viele Beispiele dafür gesehen.
- Unsere `gcd`-Funktion z. B. hatte zum Beispiel zwei Parameter `a` und `b`.
- Wir können die Funktion aufrufen, in dem wir ihre Werte in Klammern hinter den Funktionsname schreiben.

Einleitung

- Kommen wir zu einem leichteren Thema: Dem Übergeben von Argumenten an Funktionen.
- Wir haben schon viele Beispiele dafür gesehen.
- Unsere `gcd`-Funktion z. B. hatte zum Beispiel zwei Parameter `a` und `b`.
- Wir können die Funktion aufrufen, in dem wir ihre Werte in Klammern hinter den Funktionsname schreiben.
- `gcd(12, 4)` ruft `gcd` auf und weist `a` den Wert `12` und `b` den Wert `4` zu.

Einleitung

- Kommen wir zu einem leichteren Thema: Dem Übergeben von Argumenten an Funktionen.
- Wir haben schon viele Beispiele dafür gesehen.
- Unsere `gcd`-Funktion z. B. hatte zum Beispiel zwei Parameter `a` und `b`.
- Wir können die Funktion aufrufen, in dem wir ihre Werte in Klammern hinter den Funktionsname schreiben.
- `gcd(12, 4)` ruft `gcd` auf und weist `a` den Wert `12` und `b` den Wert `4` zu.
- Was können wir sonst noch mit Parametern machen?

Default Values

Default Values

- Wir können Parameter sogenannte *Default Values*, also „Standardwerte“ haben lassen.

Default Values

- Wir können Parameter sogenannte *Default Values*, also „Standardwerte“ haben lassen.
- Wenn ein Parameter einen Default Value hat, dann können wir den Parameter beim Aufrufen der Funktion weglassen.

Default Values

- Wir können Parameter sogenannte *Default Values*, also „Standardwerte“ haben lassen.
- Wenn ein Parameter einen Default Value hat, dann können wir den Parameter beim Aufrufen der Funktion weglassen.
- Wir können also einen Wert für den Parameter beim Aufrufen der Funktion angeben oder auch nicht.

Default Values

- Wir können Parameter sogenannte *Default Values*, also „Standardwerte“ haben lassen.
- Wenn ein Parameter einen Default Value hat, dann können wir den Parameter beim Aufrufen der Funktion weglassen.
- Wir können also einen Wert für den Parameter beim Aufrufen der Funktion angeben oder auch nicht.
- Im letzteren Fall bekommt der Parameter dann den Default Value.

Default Values

- Wir können Parameter sogenannte *Default Values*, also „Standardwerte“ haben lassen.
- Wenn ein Parameter einen Default Value hat, dann können wir den Parameter beim Aufrufen der Funktion weglassen.
- Wir können also einen Wert für den Parameter beim Aufrufen der Funktion angeben oder auch nicht.
- Im letzteren Fall bekommt der Parameter dann den Default Value.
- In der Funktion sieht es dann so aus, als ob wir den Default Value als Argument angegeben hätten.

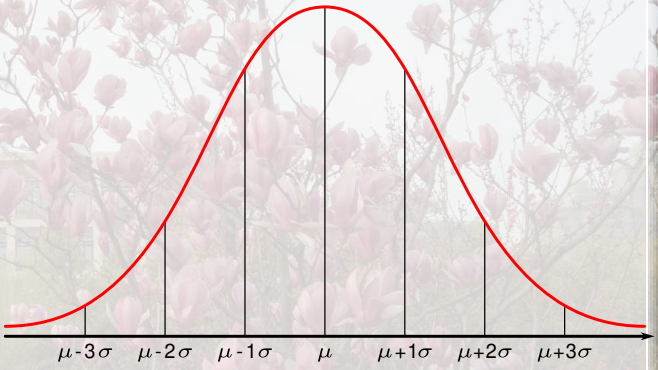
Beispiel: PDF der Normalverteilung (1)

- Als einfaches Beispiel implementieren wir die Wahrscheinlichkeitsdichte funktion f (EN: *Probability Density Function*) (PDF), der Normalverteilung[1,23,76].

Beispiel: PDF der Normalverteilung (1)

- Als einfaches Beispiel implementieren wir die Wahrscheinlichkeitsdichte funktion f (EN: *Probability Density Function*) (PDF), der Normalverteilung[1,23,76].
- Diese Funktion f ist definiert als:

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad (1)$$

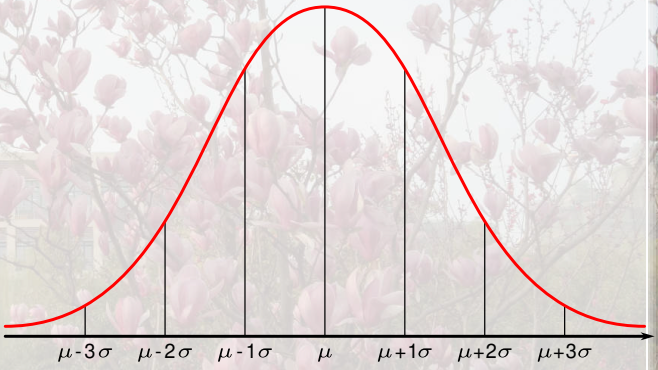


Beispiel: PDF der Normalverteilung (1)

- Als einfaches Beispiel implementieren wir die Wahrscheinlichkeitsdichte funktion f (EN: *Probability Density Function*) (PDF), der Normalverteilung[1,23,76].
- Diese Funktion f ist definiert als:

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad (1)$$

- μ ist der Erwartungswert der Verteilung, σ ist die Standardabweichung (wodurch σ^2 die Varianz ist).

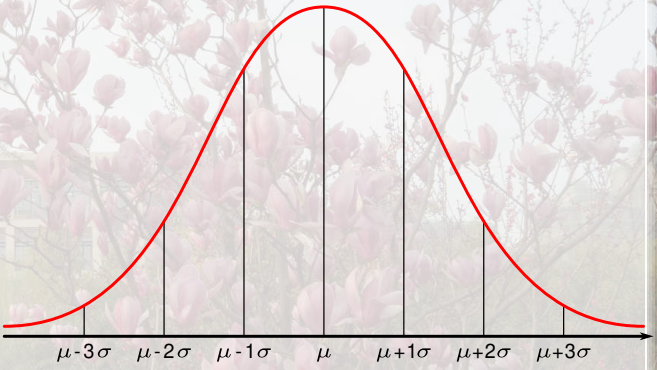


Beispiel: PDF der Normalverteilung (1)

- Diese Funktion f ist definiert als:

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad (1)$$

- μ ist der Erwartungswert der Verteilung, σ ist die Standardabweichung (wodurch σ^2 die Varianz ist).
- x ist sozusagen der Eingabewert der Funktion.

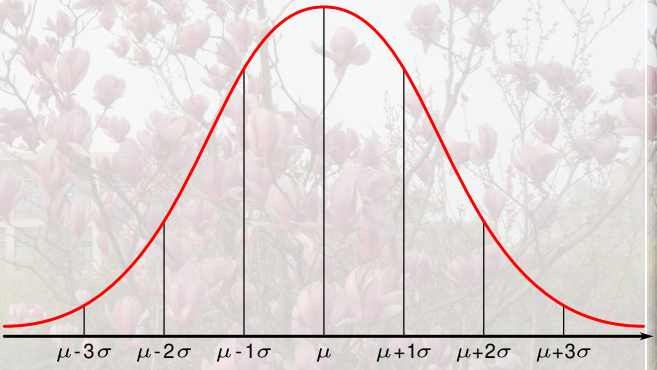


Beispiel: PDF der Normalverteilung (1)

- Diese Funktion f ist definiert als:

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad (1)$$

- μ ist der Erwartungswert der Verteilung, σ ist die Standardabweichung (wodurch σ^2 die Varianz ist).
- x ist sozusagen der Eingabewert der Funktion.
- Es reprärsentiert einen Wert, den eine normalverteilte Zufallsvariable annehmen könnte.



Beispiel: PDF der Normalverteilung (1)

- Diese Funktion f ist definiert als:

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad (1)$$

- x ist sozusagen der Eingabewert der Funktion.
- Es reprärsentiert einen Wert, den eine normalverteilte Zufallsvariable annehmen könnte.
- Diese Funktion zu implementieren ist ziemlich einfach.

```python
"""The Probability Density Function (PDF) of the Normal Distribution."""

from math import exp, pi, sqrt


def pdf(x: float, mu: float = 0.0, sigma: float = 1.0) -> float:
    """
    Compute the probability density function of the normal distribution.

    :param x: the coordinate at which to evaluate the normal PDF
    :param mu: the expected value or arithmetic mean, defaults to `0.0`.
    :param sigma: the standard deviation, defaults to `1.0`
    :return: the value of the normal PDF at `x`.
    """
    s2: float = 2 * (sigma ** 2)  # stored for reuse
    return exp((-((x - mu) ** 2)) / s2) / sqrt(pi * s2)  # compute pdf
```

Beispiel: PDF der Normalverteilung (1)

- Diese Funktion f ist definiert als:

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad (1)$$

- Es reprärsentiert einen Wert, den eine normalverteilte Zufallsvariable annehmen könnte.
- Diese Funktion zu implementieren ist ziemlich einfach.
- Die Python-Datei `normal_pdf.py` bietet eine Funktion `pdf` mit drei Parametern, nämlich `x`, `mu`, und `sigma`, die für x, μ, und σ stehen.

```python
"""The Probability Density Function (PDF) of the Normal Distribution."""

from math import exp, pi, sqrt


def pdf(x: float, mu: float = 0.0, sigma: float = 1.0) -> float:
    """
    Compute the probability density function of the normal distribution.

    :param x: the coordinate at which to evaluate the normal PDF
    :param mu: the expected value or arithmetic mean, defaults to `0.0`.
    :param sigma: the standard deviation, defaults to `1.0`
    :return: the value of the normal PDF at `x`.
    """
    s2: float = 2 * (sigma ** 2)  # stored for reuse
    return exp((-((x - mu) ** 2)) / s2) / sqrt(pi * s2)  # compute pdf
```

Beispiel: PDF der Normalverteilung (1)

- Diese Funktion f ist definiert als:

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad (1)$$

- Diese Funktion zu implementieren ist ziemlich einfach.
- Die Python-Datei `normal_pdf.py` bietet eine Funktion `pdf` mit drei Parametern, nämlich `x`, `mu`, und `sigma`, die für x, μ, und σ stehen.
- Mit den beiden parametern μ und σ von f (respektive `mu` und `sigma` von `pdf`) können wir die generelle Normalverteilung repräsentieren.

```python
"""The Probability Density Function (PDF) of the Normal Distribution."""

from math import exp, pi, sqrt


def pdf(x: float, mu: float = 0.0, sigma: float = 1.0) -> float:
    """
    Compute the probability density function of the normal distribution.

    :param x: the coordinate at which to evaluate the normal PDF
    :param mu: the expected value or arithmetic mean, defaults to `0.0`.
    :param sigma: the standard deviation, defaults to `1.0`
    :return: the value of the normal PDF at `x`.
    """
    s2: float = 2 * (sigma ** 2)  # stored for reuse
    return exp((-((x - mu) ** 2)) / s2) / sqrt(pi * s2)  # compute pdf
```

Beispiel: PDF der Normalverteilung (1)

- Diese Funktion f ist definiert als:

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad (1)$$

- Mit den beiden parametern μ und σ von f (respektive `mu` und `sigma` von `pdf`) können wir die generelle Normalverteilung repräsentieren.
- Die *Standard*normalverteilung hat $\mu = 0$ und $\sigma = 1$, ist also um den Mittelwert 0 zentriert und hat eine Standardabweichung und Varianz von 1.

```python
"""The Probability Density Function (PDF) of the Normal Distribution."""

from math import exp, pi, sqrt


def pdf(x: float, mu: float = 0.0, sigma: float = 1.0) -> float:
    """
    Compute the probability density function of the normal distribution.

    :param x: the coordinate at which to evaluate the normal PDF
    :param mu: the expected value or arithmetic mean, defaults to `0.0`.
    :param sigma: the standard deviation, defaults to `1.0`
    :return: the value of the normal PDF at `x`.
    """
    s2: float = 2 * (sigma ** 2)  # stored for reuse
    return exp((-((x - mu) ** 2)) / s2) / sqrt(pi * s2)  # compute pdf
```

Beispiel: PDF der Normalverteilung (1)

- Diese Funktion f ist definiert als:

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad (1)$$

- Die *Standard*normalverteilung hat $\mu = 0$ und $\sigma = 1$, ist also um den Mittelwert 0 zentriert und hat eine Standardabweichung und Varianz von 1.
- Sehr oft werden wir die Werte der PDF für die Standardnormalverteilung berechnen wollen.

```python
"""The Probability Density Function (PDF) of the Normal Distribution."""

from math import exp, pi, sqrt


def pdf(x: float, mu: float = 0.0, sigma: float = 1.0) -> float:
    """
    Compute the probability density function of the normal distribution.

    :param x: the coordinate at which to evaluate the normal PDF
    :param mu: the expected value or arithmetic mean, defaults to `0.0`.
    :param sigma: the standard deviation, defaults to `1.0`
    :return: the value of the normal PDF at `x`.
    """
    s2: float = 2 * (sigma ** 2)  # stored for reuse
    return exp((-((x - mu) ** 2)) / s2) / sqrt(pi * s2)  # compute pdf
```

Beispiel: PDF der Normalverteilung (1)

- Diese Funktion f ist definiert als:

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad (1)$$

- Die *Standard*normalverteilung hat $\mu = 0$ und $\sigma = 1$, ist also um den Mittelwert 0 zentriert und hat eine Standardabweichung und Varianz von 1.
- Sehr oft werden wir die Werte der PDF für die Standardnormalverteilung berechnen wollen.
- Daher definieren wir den *Default Value* von `mu` als `0.0` und den von `sigma` als `1.0`.

```python
"""The Probability Density Function (PDF) of the Normal Distribution."""

from math import exp, pi, sqrt


def pdf(x: float, mu: float = 0.0, sigma: float = 1.0) -> float:
    """
    Compute the probability density function of the normal distribution.

    :param x: the coordinate at which to evaluate the normal PDF
    :param mu: the expected value or arithmetic mean, defaults to `0.0`.
    :param sigma: the standard deviation, defaults to `1.0`
    :return: the value of the normal PDF at `x`.
    """
    s2: float = 2 * (sigma ** 2)  # stored for reuse
    return exp((-((x - mu) ** 2)) / s2) / sqrt(pi * s2)  # compute pdf
```

Beispiel: PDF der Normalverteilung (1)

- Diese Funktion f ist definiert als:

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad (1)$$

- Sehr oft werden wir die Werte der PDF für die Standardnormalverteilung berechnen wollen.
- Daher definieren wir den *Default Value* von `mu` als `0.0` und den von `sigma` als `1.0`.
- Das passiert nur im Kopf der Funktion.

```python
"""The Probability Density Function (PDF) of the Normal Distribution."""

from math import exp, pi, sqrt


def pdf(x: float, mu: float = 0.0, sigma: float = 1.0) -> float:
    """
    Compute the probability density function of the normal distribution.

    :param x: the coordinate at which to evaluate the normal PDF
    :param mu: the expected value or arithmetic mean, defaults to `0.0`.
    :param sigma: the standard deviation, defaults to `1.0`
    :return: the value of the normal PDF at `x`.
    """
    s2: float = 2 * (sigma ** 2)  # stored for reuse
    return exp((-((x - mu) ** 2)) / s2) / sqrt(pi * s2)  # compute pdf
```

Beispiel: PDF der Normalverteilung (1)

- Diese Funktion f ist definiert als:

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad (1)$$

- Daher definieren wir den *Default Value* von `mu` als `0.0` und den von `sigma` als `1.0`.
- Das passiert nur im Kopf der Funktion.
- Wir schreiben also `x: float, mu: float = 0.0, sigma: float = 1.0` anstatt von `x: float, mu: float, sigma: float`.

```python
"""The Probability Density Function (PDF) of the Normal Distribution."""

from math import exp, pi, sqrt


def pdf(x: float, mu: float = 0.0, sigma: float = 1.0) -> float:
    """
    Compute the probability density function of the normal distribution.

    :param x: the coordinate at which to evaluate the normal PDF
    :param mu: the expected value or arithmetic mean, defaults to `0.0`.
    :param sigma: the standard deviation, defaults to `1.0`
    :return: the value of the normal PDF at `x`.
    """
    s2: float = 2 * (sigma ** 2)  # stored for reuse
    return exp((-((x - mu) ** 2)) / s2) / sqrt(pi * s2)  # compute pdf
```

Beispiel: PDF der Normalverteilung (1)

- Diese Funktion f ist definiert als:

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad (1)$$

- Das passiert nur im Kopf der Funktion.
- Wir schreiben also `x: float, mu: float = 0.0, sigma: float = 1.0` anstatt von `x: float, mu: float, sigma: float`.
- Nichts ändert sich aber im Bezug darauf wie die Parameter in der Funktion verwendet werden.

```python
"""The Probability Density Function (PDF) of the Normal Distribution."""

from math import exp, pi, sqrt


def pdf(x: float, mu: float = 0.0, sigma: float = 1.0) -> float:
    """
    Compute the probability density function of the normal distribution.

    :param x: the coordinate at which to evaluate the normal PDF
    :param mu: the expected value or arithmetic mean, defaults to `0.0`.
    :param sigma: the standard deviation, defaults to `1.0`
    :return: the value of the normal PDF at `x`.
    """
    s2: float = 2 * (sigma ** 2)  # stored for reuse
    return exp((-((x - mu) ** 2)) / s2) / sqrt(pi * s2)  # compute pdf
```

Beispiel: PDF der Normalverteilung (1)

- Diese Funktion f ist definiert als:

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad (1)$$

- Wir schreiben also `x: float, mu: float = 0.0, sigma: float = 1.0` anstatt von `x: float, mu: float, sigma: float`.
- Nichts ändert sich aber im Bezug darauf wie die Parameter in der Funktion verwendet werden.
- Wir benutzen sie ganz normal, wie alle anderen Parameter auch.

```python
"""The Probability Density Function (PDF) of the Normal Distribution."""

from math import exp, pi, sqrt


def pdf(x: float, mu: float = 0.0, sigma: float = 1.0) -> float:
    """
    Compute the probability density function of the normal distribution.

    :param x: the coordinate at which to evaluate the normal PDF
    :param mu: the expected value or arithmetic mean, defaults to `0.0`.
    :param sigma: the standard deviation, defaults to `1.0`
    :return: the value of the normal PDF at `x`.
    """
    s2: float = 2 * (sigma ** 2)  # stored for reuse
    return exp((-((x - mu) ** 2)) / s2) / sqrt(pi * s2)  # compute pdf
```

Beispiel: PDF der Normalverteilung (1)

- Diese Funktion f ist definiert als:

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad (1)$$

- Nichts ändert sich aber im Bezug darauf wie die Parameter in der Funktion verwendet werden.
- Wir benutzen sie ganz normal, wie alle anderen Parameter auch.
- Der Körper der Funktion weiß nicht, wo ihre Werte herkommen.

```python
"""The Probability Density Function (PDF) of the Normal Distribution."""

from math import exp, pi, sqrt


def pdf(x: float, mu: float = 0.0, sigma: float = 1.0) -> float:
    """
    Compute the probability density function of the normal distribution.

    :param x: the coordinate at which to evaluate the normal PDF
    :param mu: the expected value or arithmetic mean, defaults to `0.0`.
    :param sigma: the standard deviation, defaults to `1.0`
    :return: the value of the normal PDF at `x`.
    """
    s2: float = 2 * (sigma ** 2)  # stored for reuse
    return exp((-((x - mu) ** 2)) / s2) / sqrt(pi * s2)  # compute pdf
```

Beispiel: PDF der Normalverteilung (1)

- Diese Funktion f ist definiert als:

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad (1)$$

- Wir benutzen sie ganz normal, wie alle anderen Parameter auch.
- Der Körper der Funktion weiß nicht, wo ihre Werte herkommen.
- Um nun die PDF der Normalverteilung zu implementieren, müssen wir erst die Funktionen `exp` und `sqrt` aus dem Modul `math` importieren, genau wie die Konstante `pi`.

```
"""The Probability Density Function (PDF) of the Normal Distribution."""

from math import exp, pi, sqrt


def pdf(x: float, mu: float = 0.0, sigma: float = 1.0) -> float:
    """
    Compute the probability density function of the normal distribution.

    :param x: the coordinate at which to evaluate the normal PDF
    :param mu: the expected value or arithmetic mean, defaults to `0.0`.
    :param sigma: the standard deviation, defaults to `1.0`
    :return: the value of the normal PDF at `x`.
    """
    s2: float = 2 * (sigma ** 2)  # stored for reuse
    return exp((-((x - mu) ** 2)) / s2) / sqrt(pi * s2)  # compute pdf
```

Beispiel: PDF der Normalverteilung (1)

- Diese Funktion f ist definiert als:

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad (1)$$

- Der Körper der Funktion weiß nicht, wo ihre Werte herkommen.
- Um nun die PDF der Normalverteilung zu implementieren, müssen wir erst die Funktionen `exp` und `sqrt` aus dem Modul `math` importieren, genau wie die Konstante `pi`.
- `exp(x)` berechnet e^x.

```python
"""The Probability Density Function (PDF) of the Normal Distribution."""

from math import exp, pi, sqrt


def pdf(x: float, mu: float = 0.0, sigma: float = 1.0) -> float:
    """
    Compute the probability density function of the normal distribution.

    :param x: the coordinate at which to evaluate the normal PDF
    :param mu: the expected value or arithmetic mean, defaults to `0.0`.
    :param sigma: the standard deviation, defaults to `1.0`
    :return: the value of the normal PDF at `x`.
    """
    s2: float = 2 * (sigma ** 2)  # stored for reuse
    return exp((-((x - mu) ** 2)) / s2) / sqrt(pi * s2)  # compute pdf
```

Beispiel: PDF der Normalverteilung (1)

- Diese Funktion f ist definiert als:

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad (1)$$

- Um nun die PDF der Normalverteilung zu implementieren, müssen wir erst die Funktionen `exp` und `sqrt` aus dem Modul `math` importieren, genau wie die Konstante `pi`.
- `exp(x)` berechnet e^x.
- Der Term $2\sigma^2$ taucht zweimal in der Gleichung auf, einmal unter der Quadratwurzel im ersten Bruch und einem im Bruch im Exponenten.

```python
"""The Probability Density Function (PDF) of the Normal Distribution."""

from math import exp, pi, sqrt


def pdf(x: float, mu: float = 0.0, sigma: float = 1.0) -> float:
    """
    Compute the probability density function of the normal distribution.

    :param x: the coordinate at which to evaluate the normal PDF
    :param mu: the expected value or arithmetic mean, defaults to `0.0`.
    :param sigma: the standard deviation, defaults to `1.0`
    :return: the value of the normal PDF at `x`.
    """
    s2: float = 2 * (sigma ** 2)  # stored for reuse
    return exp((-((x - mu) ** 2)) / s2) / sqrt(pi * s2)  # compute pdf
```

Beispiel: PDF der Normalverteilung (1)

- Diese Funktion f ist definiert als:

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad (1)$$

- `exp(x)` berechnet $e^{\mathtt{x}}$.
- Der Term $2\sigma^2$ taucht zweimal in der Gleichung auf, einmal unter der Quadratwurzel im ersten Bruch und einem im Bruch im Exponenten.
- Deshalb berechnen wir es einmal und speichern es in Variable `s2`.

```python
"""The Probability Density Function (PDF) of the Normal Distribution."""

from math import exp, pi, sqrt


def pdf(x: float, mu: float = 0.0, sigma: float = 1.0) -> float:
    """
    Compute the probability density function of the normal distribution.

    :param x: the coordinate at which to evaluate the normal PDF
    :param mu: the expected value or arithmetic mean, defaults to `0.0`.
    :param sigma: the standard deviation, defaults to `1.0`
    :return: the value of the normal PDF at `x`.
    """
    s2: float = 2 * (sigma ** 2)  # stored for reuse
    return exp((-((x - mu) ** 2)) / s2) / sqrt(pi * s2)  # compute pdf
```

Beispiel: PDF der Normalverteilung (1)

- Diese Funktion f ist definiert als:

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad (1)$$

- Der Term $2\sigma^2$ taucht zweimal in der Gleichung auf, einmal unter der Quadratwurzel im ersten Bruch und einem im Bruch im Exponenten.
- Deshalb berechnen wir es einmal und speichern es in Variable `s2`.
- Das vereifacht die Gleichung zu $[e^{-(x-\mu)^2/\texttt{s2}}]/[\sqrt{\pi * \texttt{s2}}]$.

```python
"""The Probability Density Function (PDF) of the Normal Distribution."""

from math import exp, pi, sqrt


def pdf(x: float, mu: float = 0.0, sigma: float = 1.0) -> float:
    """
    Compute the probability density function of the normal distribution.

    :param x: the coordinate at which to evaluate the normal PDF
    :param mu: the expected value or arithmetic mean, defaults to `0.0`.
    :param sigma: the standard deviation, defaults to `1.0`
    :return: the value of the normal PDF at `x`.
    """
    s2: float = 2 * (sigma ** 2)  # stored for reuse
    return exp((-((x - mu) ** 2)) / s2) / sqrt(pi * s2)  # compute pdf
```

Beispiel: PDF der Normalverteilung (1)

- Diese Funktion f ist definiert als:

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad (1)$$

- Deshalb berechnen wir es einmal und speichern es in Variable `s2`.
- Das vereifacht die Gleichung zu $[e^{-(x-\mu)^2/\texttt{s2}}]/[\sqrt{\pi * \texttt{s2}}]$.
- Beachten Sie, wie der Potenzoperator `a ** 2` äquivalent zu $\texttt{a}^2$ ist.

```python
"""The Probability Density Function (PDF) of the Normal Distribution."""

from math import exp, pi, sqrt


def pdf(x: float, mu: float = 0.0, sigma: float = 1.0) -> float:
    """
    Compute the probability density function of the normal distribution.

    :param x: the coordinate at which to evaluate the normal PDF
    :param mu: the expected value or arithmetic mean, defaults to `0.0`.
    :param sigma: the standard deviation, defaults to `1.0`
    :return: the value of the normal PDF at `x`.
    """
    s2: float = 2 * (sigma ** 2)  # stored for reuse
    return exp((-((x - mu) ** 2)) / s2) / sqrt(pi * s2)  # compute pdf
```

Beispiel: PDF der Normalverteilung (1)

- Diese Funktion f ist definiert als:

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad (1)$$

- Das vereifacht die Gleichung zu $[e^{-(x-\mu)^2/\texttt{s2}}]/[\sqrt{\pi * \texttt{s2}}]$.
- Beachten Sie, wie der Potenzoperator `a ** 2` äquivalent zu $\texttt{a}^2$ ist.
- Im Programm `use_normal_pdf.py` importieren wir nun unsere neue Funktion `pdf`.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (1)

- Diese Funktion f ist definiert als:

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad (1)$$

- Das vereifacht die Gleichung zu $[e^{-(x-\mu)^2/\texttt{s2}}]/[\sqrt{\pi * \texttt{s2}}]$.
- Beachten Sie, wie der Potenzoperator `a ** 2` äquivalent zu $\texttt{a}^2$ ist.
- Im Programm `use_normal_pdf.py` importieren wir nun unsere neue Funktion `pdf`.
- When wir `pdf` aufrufen, dann können wir die Werte für die Parameter mit Default Values weglassen.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (1)

- Diese Funktion f ist definiert als:

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \quad (1)$$

- Beachten Sie, wie der Potenzoperator `a ** 2` äquivalent zu a^2 ist.
- Im Programm `use_normal_pdf.py` importieren wir nun unsere neue Funktion `pdf`.
- When wir `pdf` aufrufen, dann können wir die Werte für die Parameter mit Default Values weglassen.
- In dem Fall bekommen diese dann ihre Default Values.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (1)

- Beachten Sie, wie der Potenzoperator `a ** 2` äquivalent zu a^2 ist.
- Im Programm `use_normal_pdf.py` importieren wir nun unsere neue Funktion `pdf`.
- When wir `pdf` aufrufen, dann können wir die Werte für die Parameter mit Default Values weglassen.
- In dem Fall bekommen diese dann ihre Default Values.
- Zum Beispiel wenn wir `pdf(0.0)` schreiben, dann ist das äquivalent zu `pdf(0.0, 0.0, 1.0)`.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (1)

- Im Programm `use_normal_pdf.py` importieren wir nun unsere neue Funktion `pdf`.
- When wir `pdf` aufrufen, dann können wir die Werte für die Parameter mit Default Values weglassen.
- In dem Fall bekommen diese dann ihre Default Values.
- Zum Beispiel wenn wir `pdf(0.0)` schreiben, dann ist das äquivalent zu `pdf(0.0, 0.0, 1.0)`.
- Wir können auch die Werte von manchen Parametern mit Default Values spezifizieren und andere weglassen.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (1)

- When wir `pdf` aufrufen, dann können wir die Werte für die Parameter mit Default Values weglassen.
- In dem Fall bekommen diese dann ihre Default Values.
- Zum Beispiel wenn wir `pdf(0.0)` schreiben, dann ist das äquivalent zu `pdf(0.0, 0.0, 1.0)`.
- Wir können auch die Werte von manchen Parametern mit Default Values spezifizieren und andere weglassen.
- Zum Beispiel ist `pdf(2.0, 3.0)` das selbe wie `pdf(2.0, 3.0, 1.0)`.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (1)

- In dem Fall bekommen diese dann ihre Default Values.
- Zum Beispiel wenn wir `pdf(0.0)` schreiben, dann ist das äquivalent zu `pdf(0.0, 0.0, 1.0)`.
- Wir können auch die Werte von manchen Parametern mit Default Values spezifizieren und andere weglassen.
- Zum Beispiel ist `pdf(2.0, 3.0)` das selbe wie `pdf(2.0, 3.0, 1.0)`.
- Wir müssen natürlich immer den Wert des ersten Parameters (`x`) spezifizieren, denn der hat keinen Default Value.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Unveränderliche Default Values

- Default Values müssen nicht unbedingt Literale sein.

Unveränderliche Default Values

- Default Values müssen nicht unbedingt Literale sein.
- Sie können das Ergebnis von beliebigen Ausdrücken sein, z. B. `sqrt(2.0)`.

Unveränderliche Default Values

- Default Values müssen nicht unbedingt Literale sein.
- Sie können das Ergebnis von beliebigen Ausdrücken sein, z. B. `sqrt(2.0)`.
- Allerdings werden sie genau nur einmal ausgewertet, nämlich wenn die Funktion definiert wird[32].

Unveränderliche Default Values

- Default Values müssen nicht unbedingt Literale sein.
- Sie können das Ergebnis von beliebigen Ausdrücken sein, z. B. `sqrt(2.0)`.
- Allerdings werden sie genau nur einmal ausgewertet, nämlich wenn die Funktion definiert wird[32].
- Sie werden dann mit dem Header der Funktion gespeichert and benutzt wann nötig.

Unveränderliche Default Values

- Default Values müssen nicht unbedingt Literale sein.
- Sie können das Ergebnis von beliebigen Ausdrücken sein, z. B. `sqrt(2.0)`.
- Allerdings werden sie genau nur einmal ausgewertet, nämlich wenn die Funktion definiert wird[32].
- Sie werden dann mit dem Header der Funktion gespeichert and benutzt wann nötig.
- Das führt zu einer interessanten Implikation: Was, wenn der Default Value für ein Argument *veränderlich* ist, z. B. eine `list` oder `set`?

Unveränderliche Default Values

- Default Values müssen nicht unbedingt Literale sein.
- Sie können das Ergebnis von beliebigen Ausdrücken sein, z. B. `sqrt(2.0)`.
- Allerdings werden sie genau nur einmal ausgewertet, nämlich wenn die Funktion definiert wird[32].
- Sie werden dann mit dem Header der Funktion gespeichert and benutzt wann nötig.
- Das führt zu einer interessanten Implikation: Was, wenn der Default Value für ein Argument *veränderlich* ist, z. B. eine `list` oder `set`?
- Nunja ... das sollten sie niemals

Unveränderliche Default Values

- Default Values müssen nicht unbedingt Literale sein.
- Sie können das Ergebnis von beliebigen Ausdrücken sein, z. B. `sqrt(2.0)`.
- Allerdings werden sie genau nur einmal ausgewertet, nämlich wenn die Funktion definiert wird[32].
- Sie werden dann mit dem Header der Funktion gespeichert and benutzt wann nötig.
- Das führt zu einer interessanten Implikation: Was, wenn der Default Value für ein Argument *veränderlich* ist, z. B. eine `list` oder `set`?
- Nunja ... das sollten sie niemals
- Das kann nämlich zu schrecklichen Problemen führen[32].

Gute Praxis

Default Values für Funktionsparameter müssen immer unveränderlich sein[32].

Unveränderliche Default Values

- Default Values müssen nicht unbedingt Literale sein.
- Sie können das Ergebnis von beliebigen Ausdrücken sein, z. B. `sqrt(2.0)`.
- Allerdings werden sie genau nur einmal ausgewertet, nämlich wenn die Funktion definiert wird[32].
- Sie werden dann mit dem Header der Funktion gespeichert and benutzt wann nötig.
- Das führt zu einer interessanten Implikation: Was, wenn der Default Value für ein Argument *veränderlich* ist, z. B. eine `list` oder `set`?
- Nunja ... das sollten sie niemals
- Das kann nämlich zu schrecklichen Problemen führen[32].
- Der Default Value für einen Funktionsparameter muss unveränderlich sein.

Unveränderliche Default Values

- Default Values müssen nicht unbedingt Literale sein.
- Sie können das Ergebnis von beliebigen Ausdrücken sein, z. B. `sqrt(2.0)`.
- Allerdings werden sie genau nur einmal ausgewertet, nämlich wenn die Funktion definiert wird[32].
- Sie werden dann mit dem Header der Funktion gespeichert and benutzt wann nötig.
- Das führt zu einer interessanten Implikation: Was, wenn der Default Value für ein Argument *veränderlich* ist, z. B. eine `list` oder `set`?
- Nunja ... das sollten sie niemals
- Das kann nämlich zu schrecklichen Problemen führen[32].
- Der Default Value für einen Funktionsparameter muss unveränderlich sein.
- Wenn Sie z. B. eine `list` verwenden, dann könnte ein Funktionsaufruf diese Liste verändern.

Unveränderliche Default Values

- Default Values müssen nicht unbedingt Literale sein.
- Sie können das Ergebnis von beliebigen Ausdrücken sein, z. B. `sqrt(2.0)`.
- Allerdings werden sie genau nur einmal ausgewertet, nämlich wenn die Funktion definiert wird[32].
- Sie werden dann mit dem Header der Funktion gespeichert and benutzt wann nötig.
- Das führt zu einer interessanten Implikation: Was, wenn der Default Value für ein Argument *veränderlich* ist, z. B. eine `list` oder `set`?
- Nunja ... das sollten sie niemals
- Das kann nämlich zu schrecklichen Problemen führen[32].
- Der Default Value für einen Funktionsparameter muss unveränderlich sein.
- Wenn Sie z. B. eine `list` verwenden, dann könnte ein Funktionsaufruf diese Liste verändern.
- Der nächste Funktionsaufruf würde dann mit der veränderten Liste arbeiten.

Unveränderliche Default Values

- Sie können das Ergebnis von beliebigen Ausdrücken sein, z. B. `sqrt(2.0)`.
- Allerdings werden sie genau nur einmal ausgewertet, nämlich wenn die Funktion definiert wird[32].
- Sie werden dann mit dem Header der Funktion gespeichert and benutzt wann nötig.
- Das führt zu einer interessanten Implikation: Was, wenn der Default Value für ein Argument *veränderlich* ist, z. B. eine `list` oder `set`?
- Nunja ... das sollten sie niemals
- Das kann nämlich zu schrecklichen Problemen führen[32].
- Der Default Value für einen Funktionsparameter muss unveränderlich sein.
- Wenn Sie z. B. eine `list` verwenden, dann könnte ein Funktionsaufruf diese Liste verändern.
- Der nächste Funktionsaufruf würde dann mit der veränderten Liste arbeiten.
- Es könnte noch schlimmer kommen: Was wenn die Funktion die Liste als Ergebnis zurückliefert?

Unveränderliche Default Values

- Allerdings werden sie genau nur einmal ausgewertet, nämlich wenn die Funktion definiert wird[32].
- Sie werden dann mit dem Header der Funktion gespeichert and benutzt wann nötig.
- Das führt zu einer interessanten Implikation: Was, wenn der Default Value für ein Argument *veränderlich* ist, z. B. eine `list` oder `set`?
- Nunja ... das sollten sie niemals
- Das kann nämlich zu schrecklichen Problemen führen[32].
- Der Default Value für einen Funktionsparameter muss unveränderlich sein.
- Wenn Sie z. B. eine `list` verwenden, dann könnte ein Funktionsaufruf diese Liste verändern.
- Der nächste Funktionsaufruf würde dann mit der veränderten Liste arbeiten.
- Es könnte noch schlimmer kommen: Was wenn die Funktion die Liste als Ergebnis zurückliefert?
- Dann könnte Kode außerhalb der Funktion den Default Value der Funktion verändern!

Unveränderliche Default Values

- Sie werden dann mit dem Header der Funktion gespeichert and benutzt wann nötig.
- Das führt zu einer interessanten Implikation: Was, wenn der Default Value für ein Argument *veränderlich* ist, z. B. eine `list` oder `set`?
- Nunja ... das sollten sie niemals
- Das kann nämlich zu schrecklichen Problemen führen[32].
- Der Default Value für einen Funktionsparameter muss unveränderlich sein.
- Wenn Sie z. B. eine `list` verwenden, dann könnte ein Funktionsaufruf diese Liste verändern.
- Der nächste Funktionsaufruf würde dann mit der veränderten Liste arbeiten.
- Es könnte noch schlimmer kommen: Was wenn die Funktion die Liste als Ergebnis zurückliefert?
- Dann könnte Kode außerhalb der Funktion den Default Value der Funktion verändern!
- Das Verhalten von solchem Kode könnte dann bliebig schwierig zu verstehen werden.

Unveränderliche Default Values

- Das führt zu einer interessanten Implikation: Was, wenn der Default Value für ein Argument *veränderlich* ist, z. B. eine `list` oder `set`?
- Nunja ... das sollten sie niemals
- Das kann nämlich zu schrecklichen Problemen führen[32].
- Der Default Value für einen Funktionsparameter muss unveränderlich sein.
- Wenn Sie z. B. eine `list` verwenden, dann könnte ein Funktionsaufruf diese Liste verändern.
- Der nächste Funktionsaufruf würde dann mit der veränderten Liste arbeiten.
- Es könnte noch schlimmer kommen: Was wenn die Funktion die Liste als Ergebnis zurückliefert?
- Dann könnte Kode außerhalb der Funktion den Default Value der Funktion verändern!
- Das Verhalten von solchem Kode könnte dann bliebig schwierig zu verstehen werden.
- Wenn wir einen Parameter mit einem veränderlichen Typ haben, dann würden wir eher `None` als Default Value benutzen.

Unveränderliche Default Values

- Nunja ... das sollten sie niemals
- Das kann nämlich zu schrecklichen Problemen führen[32].
- Der Default Value für einen Funktionsparameter muss unveränderlich sein.
- Wenn Sie z. B. eine `list` verwenden, dann könnte ein Funktionsaufruf diese Liste verändern.
- Der nächste Funktionsaufruf würde dann mit der veränderten Liste arbeiten.
- Es könnte noch schlimmer kommen: Was wenn die Funktion die Liste als Ergebnis zurückliefert?
- Dann könnte Kode außerhalb der Funktion den Default Value der Funktion verändern!
- Das Verhalten von solchem Kode könnte dann bliebig schwierig zu verstehen werden.
- Wenn wir einen Parameter mit einem veränderlichen Typ haben, dann würden wir eher `None` als Default Value benutzen.
- Wir würden dann im Funktionskörper auf `None` prüfen und entsprechendes Verhalten implementieren.

Arguments über Parametername Einspeisen

Arguments über Parametername Einspeisen

- Stellen Sie sich vor, wir haben eine Funktion `def g(x: int, y: int)`.

Arguments über Parametername Einspeisen

- Stellen Sie sich vor, wir haben eine Funktion `def g(x: int, y: int)`.
- Normalerweise würden wir sie z. B. so aufrufen `g(1, 2)`, wobei dann im Funktionskörper `x == 1` und `y == 2` gilt.

Arguments über Parametername Einspeisen

- Stellen Sie sich vor, wir haben eine Funktion `def g(x: int, y: int)`.
- Normalerweise würden wir sie z. B. so aufrufen `g(1, 2)`, wobei dann im Funktionskörper `x == 1` und `y == 2` gilt.
- Wir können aber die Argumente auch genauso definieren, wie wir Variablen zuweisen würden – in der Form `parameterName=value`.

Arguments über Parametername Einspeisen

- Stellen Sie sich vor, wir haben eine Funktion `def g(x: int, y: int)`.
- Normalerweise würden wir sie z. B. so aufrufen `g(1, 2)`, wobei dann im Funktionskörper `x == 1` und `y == 2` gilt.
- Wir können aber die Argumente auch genauso definieren, wie wir Variablen zuweisen würden – in der Form `parameterName=value`.
- Wir könnten schreiben `g(x=1, y=2)` oder, falls wir etwas freches machen wollen, `g(y=2, x=1)`.

Arguments über Parametername Einspeisen

- Stellen Sie sich vor, wir haben eine Funktion `def g(x: int, y: int)`.
- Normalerweise würden wir sie z. B. so aufrufen `g(1, 2)`, wobei dann im Funktionskörper `x == 1` und `y == 2` gilt.
- Wir können aber die Argumente auch genauso definieren, wie wir Variablen zuweisen würden – in der Form `parameterName=value`.
- Wir könnten schreiben `g(x=1, y=2)` oder, falls wir etwas freches machen wollen, `g(y=2, x=1)`.
- Beide Varianten sind äquivalent zu dem ursprünglichen Funktionsaufruf.

Arguments über Parametername Einspeisen

- Stellen Sie sich vor, wir haben eine Funktion `def g(x: int, y: int)`.
- Normalerweise würden wir sie z. B. so aufrufen `g(1, 2)`, wobei dann im Funktionskörper `x == 1` und `y == 2` gilt.
- Wir können aber die Argumente auch genauso definieren, wie wir Variablen zuweisen würden – in der Form `parameterName=value`.
- Wir könnten schreiben `g(x=1, y=2)` oder, falls wir etwas freches machen wollen, `g(y=2, x=1)`.
- Beide Varianten sind äquivalent zu dem ursprünglichen Funktionsaufruf.
- Alles, was wir gemacht haben, war explizit die Namen der Parameter anzugeben, als wir ihnen Werte zugewiesen haben.

Beispiel: PDF der Normalverteilung (2)

- Kehren wir zu unserer Funktion `pdf` zurück, die in Datei `normal_pdf.py` definiert und rechts in Program `use_normal_pdf.py` genutzt wird.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ `python3 use_normal_pdf.py` ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (2)

- Kehren wir zu unserer Funktion `pdf` zurück, die in Datei `normal_pdf.py` definiert und rechts in Program `use_normal_pdf.py` genutzt wird.
- Die Funktion `pdf` hat den `x` ohne Default Value, gefolgt von Parameter `mu` mit Default Value, welcher wiederum von Parameter `sigma` gefolgt wird, der ebenfalls einen Default Value hat.

```
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (2)

- Kehren wir zu unserer Funktion `pdf` zurück, die in Datei `normal_pdf.py` definiert und rechts in Program `use_normal_pdf.py` genutzt wird.
- Die Funktion `pdf` hat den `x` ohne Default Value, gefolgt von Parameter `mu` mit Default Value, welcher wiederum von Parameter `sigma` gefolgt wird, der ebenfalls einen Default Value hat.
- Was würden wir machen, wenn wir einen Wert für Parametre `sigma` angeben, aber `mu` bei seinem Default Value belassen wollen?

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (2)

- Kehren wir zu unserer Funktion `pdf` zurück, die in Datei `normal_pdf.py` definiert und rechts in Program `use_normal_pdf.py` genutzt wird.
- Die Funktion `pdf` hat den `x` ohne Default Value, gefolgt von Parameter `mu` mit Default Value, welcher wiederum von Parameter `sigma` gefolgt wird, der ebenfalls einen Default Value hat.
- Was würden wir machen, wenn wir einen Wert für Parametre `sigma` angeben, aber `mu` bei seinem Default Value belassen wollen?
- Wir können das, in dem wir die Werte über den Parametername spezifizieren.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (2)

- Die Funktion `pdf` hat den `x` ohne Default Value, gefolgt von Parameter `mu` mit Default Value, welcher wiederum von Parameter `sigma` gefolgt wird, der ebenfalls einen Default Value hat.
- Was würden wir machen, wenn wir einen Wert für Parametre `sigma` angeben, aber `mu` bei seinem Default Value belassen wollen?
- Wir können das, in dem wir die Werte über den Parametername spezifizieren.
- `pdf(-2.0, sigma=3.0)` übergibt `-2.0` für `x` und `3.0` für `sigma`.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (2)

- Was würden wir machen, wenn wir einen Wert für Parametre `sigma` angeben, aber `mu` bei seinem Default Value belassen wollen?
- Wir können das, in dem wir die Werte über den Parametername spezifizieren.
- `pdf(-2.0, sigma=3.0)` übergibt `-2.0` für `x` und `3.0` für `sigma`.
- Es spezifiziert keinen Wert für `mu`, wodurch dieser Parameter bei seinem Default Value bleibt.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ `python3 use_normal_pdf.py` ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (2)

- Was würden wir machen, wenn wir einen Wert für Parametre `sigma` angeben, aber `mu` bei seinem Default Value belassen wollen?
- Wir können das, in dem wir die Werte über den Parametername spezifizieren.
- `pdf(-2.0, sigma=3.0)` übergibt `-2.0` für `x` und `3.0` für `sigma`.
- Es spezifiziert keinen Wert für `mu`, wodurch dieser Parameter bei seinem Default Value bleibt.
- Der Funktionsaufruf ist daher äquivalent zu `pdf(-2.0, 0.0, 3.0)`.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (2)

- Wir können das, in dem wir die Werte über den Parametername spezifizieren.
- `pdf(-2.0, sigma=3.0)` übergibt `-2.0` für `x` und `3.0` für `sigma`.
- Es spezifiziert keinen Wert für `mu`, wodurch dieser Parameter bei seinem Default Value bleibt.
- Der Funktionsaufruf ist daher äquivalent zu `pdf(-2.0, 0.0, 3.0)`.
- Dadurch, dass wir Parameterwerte mit dem Schema `parameterName=value` übergeben können, können wir die Parameter in beliebiger Reihenfolge übergeben.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (2)

- `pdf(-2.0, sigma=3.0)` übergibt `-2.0` für `x` und `3.0` für `sigma`.
- Es spezifiziert keinen Wert für `mu`, wodurch dieser Parameter bei seinem Default Value bleibt.
- Der Funktionsaufruf ist daher äquivalent zu `pdf(-2.0, 0.0, 3.0)`.
- Dadurch, dass wir Parameterwerte mit dem Schema `parameterName=value` übergeben können, können wir die Parameter in beliebiger Reihenfolge übergeben.
- `pdf(mu=8.0, x=0.0, sigma=1.5)` ist ein Beispiel dafür.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (2)

- Es spezifiziert keinen Wert für `mu`, wodurch dieser Parameter bei seinem Default Value bleibt.
- Der Funktionsaufruf ist daher äquivalent zu `pdf(-2.0, 0.0, 3.0)`.
- Dadurch, dass wir Parameterwerte mit dem Schema `parameterName=value` übergeben können, können wir die Parameter in beliebiger Reihenfolge übergeben.
- `pdf(mu=8.0, x=0.0, sigma=1.5)` ist ein Beispiel dafür.
- Machen Sie solchen Unsinn bitte nicht.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Argumente als Dictionaries und Sequenzen

Argumente als Dictionaries und Sequenzen

- Wie wir gelernt haben, haben Parameter ja Namen.

Argumente als Dictionaries und Sequenzen

- Wie wir gelernt haben, haben Parameter ja Namen.
- Wir können sowas wie `pdf(mu=8.0, x=0.0, sigma=1.5)` schreiben, um Argumente zuzuweisen.

Argumente als Dictionaries und Sequenzen

- Wie wir gelernt haben, haben Parameter ja Namen.
- Wir können sowas wie `pdf(mu=8.0, x=0.0, sigma=1.5)` schreiben, um Argumente zuzuweisen.
- Wenn wir `pdf(-2.0, sigma=3.0)` schreiben, ist das das Selbe, als ob wir `pdf(x=-2.0, sigma=3.0)` schreiben.

Argumente als Dictionaries und Sequenzen

- Wie wir gelernt haben, haben Parameter ja Namen.
- Wir können sowas wie `pdf(mu=8.0, x=0.0, sigma=1.5)` schreiben, um Argumente zuzuweisen.
- Wenn wir `pdf(-2.0, sigma=3.0)` schreiben, ist das das Selbe, als ob wir `pdf(x=-2.0, sigma=3.0)` schreiben.
- Argumente an eine Funktion zu übergeben heißt also im Grunde, dass wir Schlüsseln (den Parameternamen) Werte zuweisen.

Argumente als Dictionaries und Sequenzen

- Wie wir gelernt haben, haben Parameter ja Namen.
- Wir können sowas wie `pdf(mu=8.0, x=0.0, sigma=1.5)` schreiben, um Argumente zuzuweisen.
- Wenn wir `pdf(-2.0, sigma=3.0)` schreiben, ist das das Selbe, als ob wir `pdf(x=-2.0, sigma=3.0)` schreiben.
- Argumente an eine Funktion zu übergeben heißt also im Grunde, dass wir Schlüsseln (den Parameternamen) Werte zuweisen.
- Das ist zumindest ein klein Wenig ähnlich zu der Art, wie wir Dictionary Literale erstellt haben.

Argumente als Dictionaries und Sequenzen

- Wie wir gelernt haben, haben Parameter ja Namen.
- Wir können sowas wie `pdf(mu=8.0, x=0.0, sigma=1.5)` schreiben, um Argumente zuzuweisen.
- Wenn wir `pdf(-2.0, sigma=3.0)` schreiben, ist das das Selbe, als ob wir `pdf(x=-2.0, sigma=3.0)` schreiben.
- Argumente an eine Funktion zu übergeben heißt also im Grunde, dass wir Schlüsseln (den Parameternamen) Werte zuweisen.
- Das ist zumindest ein klein Wenig ähnlich zu der Art, wie wir Dictionary Literale erstellt haben.
- Tatsächlich erlaubt uns Python, die Argumente von Funktionsaufrufen als Kollektionen zu konstruieren.

Argumente als Dictionaries und Sequenzen

- Wie wir gelernt haben, haben Parameter ja Namen.
- Wir können sowas wie `pdf(mu=8.0, x=0.0, sigma=1.5)` schreiben, um Argumente zuzuweisen.
- Wenn wir `pdf(-2.0, sigma=3.0)` schreiben, ist das das Selbe, als ob wir `pdf(x=-2.0, sigma=3.0)` schreiben.
- Argumente an eine Funktion zu übergeben heißt also im Grunde, dass wir Schlüsseln (den Parameternamen) Werte zuweisen.
- Das ist zumindest ein klein Wenig ähnlich zu der Art, wie wir Dictionary Literale erstellt haben.
- Tatsächlich erlaubt uns Python, die Argumente von Funktionsaufrufen als Kollektionen zu konstruieren.
- Klingt komisch, aber schauen wir uns das mal an.

Beispiel: PDF der Normalverteilung (3)

- Tatsächlich erlaubt uns Python, die Argumente von Funktionsaufrufen als Kollektionen zu konstruieren.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Tatsächlich erlaubt uns Python, die Argumente von Funktionsaufrufen als Kollektionen zu konstruieren.
- Wir können ein Dictionary wmit den `{"x": -2.0, "sigma": 3.0}` erstellen.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Tatsächlich erlaubt uns Python, die Argumente von Funktionsaufrufen als Kollektionen zu konstruieren.
- Wir können ein Dictionary wmit den `{"x": -2.0, "sigma": 3.0}` erstellen.
- Speichern wir dieses Dictionary in der Variable `args_dict`.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Tatsächlich erlaubt uns Python, die Argumente von Funktionsaufrufen als Kollektionen zu konstruieren.
- Wir können ein Dictionary wmit den `{"x": -2.0, "sigma": 3.0}` erstellen.
- Speichern wir dieses Dictionary in der Variable `args_dict`.
- Können wir nun die Schlüssel-Wert-Paare aus `args_dict` als Parameter-Argument-Paare on `pdf` übergeben?

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Tatsächlich erlaubt uns Python, die Argumente von Funktionsaufrufen als Kollektionen zu konstruieren.
- Wir können ein Dictionary wmit den `{"x": -2.0, "sigma": 3.0}` erstellen.
- Speichern wir dieses Dictionary in der Variable `args_dict`.
- Können wir nun die Schlüssel-Wert-Paare aus `args_dict` als Parameter-Argument-Paare on `pdf` übergeben?
- Ja, das geht.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Wir können ein Dictionary wmit den `{"x": -2.0, "sigma": 3.0}` erstellen.
- Speichern wir dieses Dictionary in der Variable `args_dict`.
- Können wir nun die Schlüssel-Wert-Paare aus `args_dict` als Parameter-Argument-Paare on `pdf` übergeben?
- Ja, das geht.
- Wir müssen nur `pdf(**args_dict)` schreiben.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Speichern wir dieses Dictionary in der Variable `args_dict`.
- Können wir nun die Schlüssel-Wert-Paare aus `args_dict` als Parameter-Argument-Paare on `pdf` übergeben?
- Ja, das geht.
- Wir müssen nur `pdf(**args_dict)` schreiben.
- Wenn wir das machen, dann wird das Dictionary `args_dict` „ausgepackt“ und alle seine Werte unter ihren Namen als Argumente für die entsprechenden Parameter übergeben.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ `python3 use_normal_pdf.py` ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Können wir nun die Schlüssel-Wert-Paare aus `args_dict` als Parameter-Argument-Paare on `pdf` übergeben?
- Ja, das geht.
- Wir müssen nur `pdf(**args_dict)` schreiben.
- Wenn wir das machen, dann wird das Dictionary `args_dict` „ausgepackt“ und alle seine Werte unter ihren Namen als Argumente für die entsprechenden Parameter übergeben.
- `pdf(**args_dict)` ist also das selbe wie `pdf(x=-2.0, sigma=3.0)`.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Ja, das geht.
- Wir müssen nur `pdf(**args_dict)` schreiben.
- Wenn wir das machen, dann wird das Dictionary `args_dict` „ausgepackt“ und alle seine Werte unter ihren Namen als Argumente für die entsprechenden Parameter übergeben.
- `pdf(**args_dict)` ist also das selbe wie `pdf(x=-2.0, sigma=3.0)`.
- Hier gibt es zwei Dinge, die zu beachten sind.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ `python3 use_normal_pdf.py` ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Wir müssen nur `pdf(**args_dict)` schreiben.
- Wenn wir das machen, dann wird das Dictionary `args_dict` „ausgepackt“ und alle seine Werte unter ihren Namen als Argumente für die entsprechenden Parameter übergeben.
- `pdf(**args_dict)` ist also das selbe wie `pdf(x=-2.0, sigma=3.0)`.
- Hier gibt es zwei Dinge, die zu beachten sind:
- Erstens der Doppel-Stern `**`.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ `python3 use_normal_pdf.py` ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Wenn wir das machen, dann wird das Dictionary `args_dict` „ausgepackt“ und alle seine Werte unter ihren Namen als Argumente für die entsprechenden Parameter übergeben.
- `pdf(**args_dict)` ist also das selbe wie `pdf(x=-2.0, sigma=3.0)`.
- Hier gibt es zwei Dinge, die zu beachten sind:
- Erstens der Doppel-Stern `**`.
- Sterne werden hier auch „Wildcard“ oder „Asterisk“ genannt.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- `pdf(**args_dict)` ist also das selbe wie `pdf(x=-2.0, sigma=3.0)`.
- Hier gibt es zwei Dinge, die zu beachten sind:
- Erstens der Doppel-Stern `**`.
- Sterne werden hier auch „Wildcard“ oder „Asterisk“ genannt.
- Der Doppel-Stern `**` wird vor das Dictionary geschrieben.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ `python3 use_normal_pdf.py` ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Hier gibt es zwei Dinge, die zu beachten sind:
- Erstens der Doppel-Stern `**`.
- Sterne werden hier auch „Wildcard“ oder „Asterisk“ genannt.
- Der Doppel-Stern `**` wird vor das Dictionary geschrieben.
- Er sagt Python, dass das Dictionary ausgepackt und zum Füllen der Argumentliste verwendet werden soll.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ `python3 use_normal_pdf.py` ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Erstens der Doppel-Stern **.
- Sterne werden hier auch „Wildcard“ oder „Asterisk“ genannt.
- Der Doppel-Stern ** wird vor das Dictionary geschrieben.
- Er sagt Python, dass das Dictionary ausgepackt und zum Füllen der Argumentliste verwendet werden soll.
- Zweitens gelten Default Values auch hier.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Sterne werden hier auch „Wildcard“ oder „Asterisk“ genannt.
- Der Doppel-Stern `**` wird vor das Dictionary geschrieben.
- Er sagt Python, dass das Dictionary ausgepackt und zum Füllen der Argumentliste verwendet werden soll.
- Zweitens gelten Default Values auch hier.
- Wir haben keinen Wert für `mu` angegeben.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ `python3 use_normal_pdf.py` ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Der Doppel-Stern `**` wird vor das Dictionary geschrieben.
- Er sagt Python, dass das Dictionary ausgepackt und zum Füllen der Argumentliste verwendet werden soll.
- Zweitens gelten Default Values auch hier.
- Wir haben keinen Wert für `mu` angegeben.
- Daher wird `mu` in dem Funktionsaufruf seinen Default Wert `0.0` haben.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Er sagt Python, dass das Dictionary ausgepackt und zum Füllen der Argumentliste verwendet werden soll.
- Zweitens gelten Default Values auch hier.
- Wir haben keinen Wert für `mu` angegeben.
- Daher wird `mu` in dem Funktionsaufruf seinen Default Wert `0.0` haben.
- Vielleicht wollen wir die Argumente aber nicht basierend auf den Parameternamen eingeben, sondern basierend auf ihrer Position.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Zweitens gelten Default Values auch hier.
- Wir haben keinen Wert für `mu` angegeben.
- Daher wird `mu` in dem Funktionsaufruf seinen Default Wert `0.0` haben.
- Vielleicht wollen wir die Argumente aber nicht basierend auf den Parameternamen eingeben, sondern basierend auf ihrer Position.
- So haben wir es ja bisher immer gemacht.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Wir haben keinen Wert für `mu` angegeben.
- Daher wird `mu` in dem Funktionsaufruf seinen Default Wert `0.0` haben.
- Vielleicht wollen wir die Argumente aber nicht basierend auf den Parameternamen eingeben, sondern basierend auf ihrer Position.
- So haben wir es ja bisher immer gemacht.
- Dann können wir eine Sequenz, z. B. eine `list` oder ein `tuple` mit den Parameterwerten konstruieren.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ `python3 use_normal_pdf.py` ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Daher wird `mu` in dem Funktionsaufruf seinen Default Wert `0.0` haben.
- Vielleicht wollen wir die Argumente aber nicht basierend auf den Parameternamen eingeben, sondern basierend auf ihrer Position.
- So haben wir es ja bisher immer gemacht.
- Dann können wir eine Sequenz, z. B. eine `list` oder ein `tuple` mit den Parameterwerten konstruieren.
- Natürlich speichern `lists` und `tuples` keine Schlüssel-Wert-Beziehungen, sondern nur Werte and Positionen.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ `python3 use_normal_pdf.py` ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Vielleicht wollen wir die Argumente aber nicht basierend auf den Parameternamen eingeben, sondern basierend auf ihrer Position.
- So haben wir es ja bisher immer gemacht.
- Dann können wir eine Sequenz, z. B. eine `list` oder ein `tuple` mit den Parameterwerten konstruieren.
- Natürlich speichern `lists` und `tuples` keine Schlüssel-Wert-Beziehungen, sondern nur Werte and Positionen.
- Wir könnten ein Tupel `args_tuple` mit den Werten `(-2.0, 7.0, 3.0)` konstruieren.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ `python3 use_normal_pdf.py` ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- So haben wir es ja bisher immer gemacht.
- Dann können wir eine Sequenz, z. B. eine `list` oder ein `tuple` mit den Parameterwerten konstruieren.
- Natürlich speichern `lists` und `tuples` keine Schlüssel-Wert-Beziehungen, sondern nur Werte and Positionen.
- Wir könnten ein Tupel `args_tuple` mit den Werten `(-2.0, 7.0, 3.0)` konstruieren.
- Dann rufen wir `pdf(*args_tuple)` auf.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ `python3 use_normal_pdf.py` ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Dann können wir eine Sequenz, z. B. eine `list` oder ein `tuple` mit den Parameterwerten konstruieren.
- Natürlich speichern `lists` und `tuples` keine Schlüssel-Wert-Beziehungen, sondern nur Werte and Positionen.
- Wir könnten ein Tupel `args_tuple` mit den Werten `(-2.0, 7.0, 3.0)` konstruieren.
- Dann rufen wir `pdf(*args_tuple)` auf.
- Das füllt die Werte aus dem Tupel einen nach dem anderen in die Parameter der Funktion ein.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Natürlich speichern `lists` und `tuples` keine Schlüssel-Wert-Beziehungen, sondern nur Werte and Positionen.
- Wir könnten ein Tupel `args_tuple` mit den Werten `(-2.0, 7.0, 3.0)` konstruieren.
- Dann rufen wir `pdf(*args_tuple)` auf.
- Das füllt die Werte aus dem Tupel einen nach dem anderen in die Parameter der Funktion ein.
- Es ist im Grunde äquivalent zu `pdf(-2.0, 7.0, 3.0)`.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Wir könnten ein Tupel `args_tuple` mit den Werten `(-2.0, 7.0, 3.0)` konstruieren.
- Dann rufen wir `pdf(*args_tuple)` auf.
- Das füllt die Werte aus dem Tupel einen nach dem anderen in die Parameter der Funktion ein.
- Es ist im Grunde äquivalent zu `pdf(-2.0, 7.0, 3.0)`.
- Dieses Mal schreiben wir nur einen einzigen „Wildcard“ `*` vor `args_tuple`.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ `python3 use_normal_pdf.py` ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Dann rufen wir `pdf(*args_tuple)` auf.
- Das füllt die Werte aus dem Tupel einen nach dem anderen in die Parameter der Funktion ein.
- Es ist im Grunde äquivalent zu `pdf(-2.0, 7.0, 3.0)`.
- Dieses Mal schreiben wir nur einen einzigen „Wildcard“ `*` vor `args_tuple`.
- Genauso gut können wir die Parameterwerte durch das „Auspacken“ einer Liste einfüllen.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ `python3 use_normal_pdf.py` ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Das füllt die Werte aus dem Tupel einen nach dem anderen in die Parameter der Funktion ein.
- Es ist im Grunde äquivalent zu `pdf(-2.0, 7.0, 3.0)`.
- Dieses Mal schreiben wir nur einen einzigen „Wildcard“ `*` vor `args_tuple`.
- Genauso gut können wir die Parameterwerte durch das „Auspacken“ einer Liste einfüllen.
- Wir erstellen dafür die Liste `args_list = [2.0, 3.0]`.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Es ist im Grunde äquivalent zu `pdf(-2.0, 7.0, 3.0)`.
- Dieses Mal schreiben wir nur einen einzigen „Wildcard“ `*` vor `args_tuple`.
- Genauso gut können wir die Parameterwerte durch das „Auspacken“ einer Liste einfüllen.
- Wir erstellen dafür die Liste `args_list = [2.0, 3.0]`.
- Der Aufruf `pdf(*args_list)` ist das selbe, als wenn wir `pdf(2.0, 3.0)` schreiben würden.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ `python3 use_normal_pdf.py` ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Dieses Mal schreiben wir nur einen einzigen „Wildcard“ `*` vor `args_tuple`.
- Genauso gut können wir die Parameterwerte durch das „Auspacken“ einer Liste einfüllen.
- Wir erstellen dafür die Liste `args_list = [2.0, 3.0]`.
- Der Aufruf `pdf(*args_list)` ist das selbe, als wenn wir `pdf(2.0, 3.0)` schreiben würden.
- Das ist wiederum das selbe wie `pdf(2.0, 3.0, 1.0)`.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Beispiel: PDF der Normalverteilung (3)

- Genauso gut können wir die Parameterwerte durch das „Auspacken“ einer Liste einfüllen.
- Wir erstellen dafür die Liste `args_list = [2.0, 3.0]`.
- Der Aufruf `pdf(*args_list)` ist das selbe, als wenn wir `pdf(2.0, 3.0)` schreiben würden.
- Das ist wiederum das selbe wie `pdf(2.0, 3.0, 1.0)`.
- Denn die Default Werte müssen nach wie vor nicht explizit hingeschrieben werden.

```python
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ python3 use_normal_pdf.py ↓

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

Zusammenfassung

Wozu das alles?

- Auf den ersten Blick sieht das alles nicht so nützlich aus.

Wozu das alles?

- Auf den ersten Blick sieht das alles nicht so nützlich aus.
- Wozu brauchen wir Default Values?

Wozu das alles?

- Auf den ersten Blick sieht das alles nicht so nützlich aus.
- Wozu brauchen wir Default Values?
- In manchen Fällen möchte man es den Benutzern ermöglichen, eine Funktion zu „anzupassen“.

Wozu das alles?

- Auf den ersten Blick sieht das alles nicht so nützlich aus.
- Wozu brauchen wir Default Values?
- In manchen Fällen möchte man es den Benutzern ermöglichen, eine Funktion zu „anzupassen“.
- Ein typisches Beispiel ist die `plot`-Methode des `Axes`-Objektes von der populären Matplotlib library.

Wozu das alles?

- Auf den ersten Blick sieht das alles nicht so nützlich aus.
- Wozu brauchen wir Default Values?
- In manchen Fällen möchte man es den Benutzern ermöglichen, eine Funktion zu „anzupassen“.
- Ein typisches Beispiel ist die `plot`-Methode des `Axes`-Objektes von der populären Matplotlib library.
- Normalerweise braucht man nur Sequenzen von x- und y-Koordinaten an diese Funktion zu übergeben, und sie wird eine Linie malen, die durch alle spezifizierten Punkte geht.

Wozu das alles?

- Auf den ersten Blick sieht das alles nicht so nützlich aus.
- Wozu brauchen wir Default Values?
- In manchen Fällen möchte man es den Benutzern ermöglichen, eine Funktion zu „anzupassen“.
- Ein typisches Beispiel ist die `plot`-Methode des `Axes`-Objektes von der populären Matplotlib library.
- Normalerweise braucht man nur Sequenzen von x- und y-Koordinaten an diese Funktion zu übergeben, und sie wird eine Linie malen, die durch alle spezifizierten Punkte geht.
- Man kann aber noch optional Farben für die Line, Markierungen die an den Punkten gemalt werden sollen, eine Strich-Stil für die Line, ein Label, Farben und Größen für die Markierungen, eine z-Reihenfolge, für den Fall, dass mehrere Linien gemalt werden sollen, und so weiter, angeben.

Wozu das alles?

- Auf den ersten Blick sieht das alles nicht so nützlich aus.
- Wozu brauchen wir Default Values?
- In manchen Fällen möchte man es den Benutzern ermöglichen, eine Funktion zu „anzupassen“.
- Ein typisches Beispiel ist die `plot`-Methode des `Axes`-Objektes von der populären Matplotlib library.
- Normalerweise braucht man nur Sequenzen von x- und y-Koordinaten an diese Funktion zu übergeben, und sie wird eine Linie malen, die durch alle spezifizierten Punkte geht.
- Man kann aber noch optional Farben für die Line, Markierungen die an den Punkten gemalt werden sollen, eine Strich-Stil für die Line, ein Label, Farben und Größen für die Markierungen, eine z-Reihenfolge, für den Fall, dass mehrere Linien gemalt werden sollen, und so weiter, angeben.
- Durch die Default Values wird der Funktionsaufruf in den meisten Fällen sehr einfach, während komplexe Formatierungen trotzdem möglich sind.

Wozu das alles?

- Auf den ersten Blick sieht das alles nicht so nützlich aus.
- Wozu brauchen wir Default Values?
- In manchen Fällen möchte man es den Benutzern ermöglichen, eine Funktion zu „anzupassen“.
- Ein typisches Beispiel ist die `plot`-Methode des `Axes`-Objektes von der populären Matplotlib library.
- Normalerweise braucht man nur Sequenzen von x- und y-Koordinaten an diese Funktion zu übergeben, und sie wird eine Linie malen, die durch alle spezifizierten Punkte geht.
- Man kann aber noch optional Farben für die Line, Markierungen die an den Punkten gemalt werden sollen, eine Strich-Stil für die Line, ein Label, Farben und Größen für die Markierungen, eine z-Reihenfolge, für den Fall, dass mehrere Linien gemalt werden sollen, und so weiter, angeben.
- Durch die Default Values wird der Funktionsaufruf in den meisten Fällen sehr einfach, während komplexe Formatierungen trotzdem möglich sind.
- Aus diesem Beispiel können wir auch den Use Case für das Zusammenbauen von Argumenten mit Hilfe einer Kollektion ableiten.

Wozu das alles?

- In manchen Fällen möchte man es den Benutzern ermöglichen, eine Funktion zu „anzupassen“.
- Ein typisches Beispiel ist die `plot`-Methode des `Axes`-Objektes von der populären Matplotlib library.
- Normalerweise braucht man nur Sequenzen von x- und y-Koordinaten an diese Funktion zu übergeben, und sie wird eine Linie malen, die durch alle spezifizierten Punkte geht.
- Man kann aber noch optional Farben für die Line, Markierungen die an den Punkten gemalt werden sollen, eine Strich-Stil für die Line, ein Label, Farben und Größen für die Markierungen, eine z-Reihenfolge, für den Fall, dass mehrere Linien gemalt werden sollen, und so weiter, angeben.
- Durch die Default Values wird der Funktionsaufruf in den meisten Fällen sehr einfach, während komplexe Formatierungen trotzdem möglich sind.
- Aus diesem Beispiel können wir auch den Use Case für das Zusammenbauen von Argumenten mit Hilfe einer Kollektion ableiten.
- Stellen wir uns vor, dass wir eine eigene Funktion zum Malen mit Hilfe von Matplotlib entwickeln.

Wozu das alles?

- Ein typisches Beispiel ist die `plot`-Methode des `Axes`-Objektes von der populären Matplotlib library.
- Normalerweise braucht man nur Sequenzen von x- und y-Koordinaten an diese Funktion zu übergeben, und sie wird eine Linie malen, die durch alle spezifizierten Punkte geht.
- Man kann aber noch optional Farben für die Line, Markierungen die an den Punkten gemalt werden sollen, eine Strich-Stil für die Line, ein Label, Farben und Größen für die Markierungen, eine z-Reihenfolge, für den Fall, dass mehrere Linien gemalt werden sollen, und so weiter, angeben.
- Durch die Default Values wird der Funktionsaufruf in den meisten Fällen sehr einfach, während komplexe Formatierungen trotzdem möglich sind.
- Aus diesem Beispiel können wir auch den Use Case für das Zusammenbauen von Argumenten mit Hilfe einer Kollektion ableiten.
- Stellen wir uns vor, dass wir eine eigene Funktion zum Malen mit Hilfe von Matplotlib entwickeln.
- Sagen wir, unsere Funktion macht so einen Matplotlib Plot-Aufruf mit zehn Parametern.

Wozu das alles?

- Normalerweise braucht man nur Sequenzen von x- und y-Koordinaten an diese Funktion zu übergeben, und sie wird eine Linie malen, die durch alle spezifizierten Punkte geht.
- Man kann aber noch optional Farben für die Line, Markierungen die an den Punkten gemalt werden sollen, eine Strich-Stil für die Line, ein Label, Farben und Größen für die Markierungen, eine z-Reihenfolge, für den Fall, dass mehrere Linien gemalt werden sollen, und so weiter, angeben.
- Durch die Default Values wird der Funktionsaufruf in den meisten Fällen sehr einfach, während komplexe Formatierungen trotzdem möglich sind.
- Aus diesem Beispiel können wir auch den Use Case für das Zusammenbauen von Argumenten mit Hilfe einer Kollektion ableiten.
- Stellen wir uns vor, dass wir eine eigene Funktion zum Malen mit Hilfe von Matplotlib entwickeln.
- Sagen wir, unsere Funktion macht so einen Matplotlib Plot-Aufruf mit zehn Parametern.
- Aber wir haben einen Sonderfall, in dem wir noch einen zusätzlichen Parameter, sagen wir einen Strich-Stil mit angeben wollen.

Wozu das alles?

- Man kann aber noch optional Farben für die Line, Markierungen die an den Punkten gemalt werden sollen, eine Strich-Stil für die Line, ein Label, Farben und Größen für die Markierungen, eine z-Reihenfolge, für den Fall, dass mehrere Linien gemalt werden sollen, und so weiter, angeben.
- Durch die Default Values wird der Funktionsaufruf in den meisten Fällen sehr einfach, während komplexe Formatierungen trotzdem möglich sind.
- Aus diesem Beispiel können wir auch den Use Case für das Zusammenbauen von Argumenten mit Hilfe einer Kollektion ableiten.
- Stellen wir uns vor, dass wir eine eigene Funktion zum Malen mit Hilfe von Matplotlib entwickeln.
- Sagen wir, unsere Funktion macht so einen Matplotlib Plot-Aufruf mit zehn Parametern.
- Aber wir haben einen Sonderfall, in dem wir noch einen zusätzlichen Parameter, sagen wir einen Strich-Stil mit angeben wollen.
- Dann könnten wir ein `if ... else` in unserem Kode haben, dessen einer Zwei den zehn-Parameter-Aufruf und dessen anderer Zwei den elf-Parameter-Aufruf macht.

Wozu das alles?

- Durch die Default Values wird der Funktionsaufruf in den meisten Fällen sehr einfach, während komplexe Formatierungen trotzdem möglich sind.
- Aus diesem Beispiel können wir auch den Use Case für das Zusammenbauen von Argumenten mit Hilfe einer Kollektion ableiten.
- Stellen wir uns vor, dass wir eine eigene Funktion zum Malen mit Hilfe von Matplotlib entwickeln.
- Sagen wir, unsere Funktion macht so einen Matplotlib Plot-Aufruf mit zehn Parametern.
- Aber wir haben einen Sonderfall, in dem wir noch einen zusätzlichen Parameter, sagen wir einen Strich-Stil mit angeben wollen.
- Dann könnten wir ein `if ... else` in unserem Kode haben, dessen einer Zwei den zehn-Parameter-Aufruf und dessen anderer Zwei den elf-Parameter-Aufruf macht.
- Dadurch haben wir dann einen sehr komplexen Funktionsaufruf, der im Grunde zweimal fast gleich auftaucht.

Wozu das alles?

- Durch die Default Values wird der Funktionsaufruf in den meisten Fällen sehr einfach, während komplexe Formatierungen trotzdem möglich sind.
- Aus diesem Beispiel können wir auch den Use Case für das Zusammenbauen von Argumenten mit Hilfe einer Kollektion ableiten.
- Stellen wir uns vor, dass wir eine eigene Funktion zum Malen mit Hilfe von Matplotlib entwickeln.
- Sagen wir, unsere Funktion macht so einen Matplotlib Plot-Aufruf mit zehn Parametern.
- Aber wir haben einen Sonderfall, in dem wir noch einen zusätzlichen Parameter, sagen wir einen Strich-Stil mit angeben wollen.
- Dann könnten wir ein `if ... else` in unserem Kode haben, dessen einer Zwei den zehn-Parameter-Aufruf und dessen anderer Zwei den elf-Parameter-Aufruf macht.
- Dadurch haben wir dann einen sehr komplexen Funktionsaufruf, der im Grunde zweimal fast gleich auftaucht.
- Stattdessen könnte man ein `dict` mit den zehn Parametern konstruieren.

Wozu das alles?

- Aus diesem Beispiel können wir auch den Use Case für das Zusammenbauen von Argumenten mit Hilfe einer Kollektion ableiten.
- Stellen wir uns vor, dass wir eine eigene Funktion zum Malen mit Hilfe von Matplotlib entwickeln.
- Sagen wir, unsere Funktion macht so einen Matplotlib Plot-Aufruf mit zehn Parametern.
- Aber wir haben einen Sonderfall, in dem wir noch einen zusätzlichen Parameter, sagen wir einen Strich-Stil mit angeben wollen.
- Dann könnten wir ein `if ... else` in unserem Kode haben, dessen einer Zwei den zehn-Parameter-Aufruf und dessen anderer Zwei den elf-Parameter-Aufruf macht.
- Dadurch haben wir dann einen sehr komplexen Funktionsaufruf, der im Grunde zweimal fast gleich auftaucht.
- Stattdessen könnte man ein `dict` mit den zehn Parametern konstruieren.
- Im `if` könnten wir dann den elften Parameter hinzufügen, wenn nötig.

Wozu das alles?

- Stellen wir uns vor, dass wir eine eigene Funktion zum Malen mit Hilfe von Matplotlib entwickeln.
- Sagen wir, unsere Funktion macht so einen Matplotlib Plot-Aufruf mit zehn Parametern.
- Aber wir haben einen Sonderfall, in dem wir noch einen zusätzlichen Parameter, sagen wir einen Strich-Stil mit angeben wollen.
- Dann könnten wir ein `if ... else` in unserem Kode haben, dessen einer Zwei den zehn-Parameter-Aufruf und dessen anderer Zwei den elf-Parameter-Aufruf macht.
- Dadurch haben wir dann einen sehr komplexen Funktionsaufruf, der im Grunde zweimal fast gleich auftaucht.
- Stattdessen könnte man ein `dict` mit den zehn Parametern konstruieren.
- Im `if` könnten wir dann den elften Parameter hinzufügen, wenn nötig.
- Dann brauchen wir nur einen Funktionsaufruf, eben mit der Doppel-Wildcard-Methode.

Wozu das alles?

- Sagen wir, unsere Funktion macht so einen Matplotlib Plot-Aufruf mit zehn Parametern.
- Aber wir haben einen Sonderfall, in dem wir noch einen zusätzlichen Parameter, sagen wir einen Strich-Stil mit angeben wollen.
- Dann könnten wir ein `if ... else` in unserem Kode haben, dessen einer Zwei den zehn-Parameter-Aufruf und dessen anderer Zwei den elf-Parameter-Aufruf macht.
- Dadurch haben wir dann einen sehr komplexen Funktionsaufruf, der im Grunde zweimal fast gleich auftaucht.
- Stattdessen könnte man ein `dict` mit den zehn Parametern konstruieren.
- Im `if` könnten wir dann den elften Parameter hinzufügen, wenn nötig.
- Dann brauchen wir nur einen Funktionsaufruf, eben mit der Doppel-Wildcard-Methode.
- Der Kode ist viel kürzer.

Wozu das alles?

- Aber wir haben einen Sonderfall, in dem wir noch einen zusätzlichen Parameter, sagen wir einen Strich-Stil mit angeben wollen.
- Dann könnten wir ein `if ... else` in unserem Kode haben, dessen einer Zwei den zehn-Parameter-Aufruf und dessen anderer Zwei den elf-Parameter-Aufruf macht.
- Dadurch haben wir dann einen sehr komplexen Funktionsaufruf, der im Grunde zweimal fast gleich auftaucht.
- Stattdessen könnte man ein `dict` mit den zehn Parametern konstruieren.
- Im `if` könnten wir dann den elften Parameter hinzufügen, wenn nötig.
- Dann brauchen wir nur einen Funktionsaufruf, eben mit der Doppel-Wildcard-Methode.
- Der Kode ist viel kürzer.
- Der Unterschied zwischen beiden Fällen ist viel klarer.

Wozu das alles?

- Dann könnten wir ein `if ... else` in unserem Kode haben, dessen einer Zwei den zehn-Parameter-Aufruf und dessen anderer Zwei den elf-Parameter-Aufruf macht.
- Dadurch haben wir dann einen sehr komplexen Funktionsaufruf, der im Grunde zweimal fast gleich auftaucht.
- Stattdessen könnte man ein `dict` mit den zehn Parametern konstruieren.
- Im `if` könnten wir dann den elften Parameter hinzufügen, wenn nötig.
- Dann brauchen wir nur einen Funktionsaufruf, eben mit der Doppel-Wildcard-Methode.
- Der Kode ist viel kürzer.
- Der Unterschied zwischen beiden Fällen ist viel klarer.
- Die Chance, Fehler zu machen ist auch viel kleiner.

Zusammenfassung

- Mit Default Values und dem Funktionsaufruf vie `*` oder `**` haben wir zwei weitere Aspekte kennengelernt die uns das Arbeiten mit Funktionen in Python erleichtern.

Zusammenfassung

- Mit Default Values und dem Funktionsaufruf vie `*` oder `**` haben wir zwei weitere Aspekte kennengelernent die uns das Arbeiten mit Funktionen in Python erleichtern.
- Mit den Default Values können wir flexible funktionsbasierte APIs entwickeln, bei denen die Benutzer die Werte für einige Parameter angeben und andere auf vernünftigen Voreinstellungen belassen können.

Zusammenfassung

- Mit Default Values und dem Funktionsaufruf vie `*` oder `**` haben wir zwei weitere Aspekte kennengelernent die uns das Arbeiten mit Funktionen in Python erleichtern.
- Mit den Default Values können wir flexible funktionsbasierte APIs entwickeln, bei denen die Benutzer die Werte für einige Parameter angeben und andere auf vernünftigen Voreinstellungen belassen können.
- Das Verwenden der `*`- oder `**`-Methode zum Funktionsaufruf mit Kollektionen von Argumenten erlaubt es uns, einfacheren Kode zu schreiben wenn Funktionen mit vielen Parametern auf mehrfach leicht verschiedene Art aufgerufen werden sollen.

谢谢您们！

Thank you!

Vielen Dank!

References I

[1] Milton Abramowitz und Irene A. Stegun, Hrsg. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Tenth Printing, with corrections. Bd. 55 der Reihe National Bureau of Standards Applied Mathematics Series. Gaithersburg, MD, USA: United States Department of Commerce, National Bureau of Standards und Washington, D.C., USA: United States Government Printing Office, Juni 1964–Dez. 1972. ISSN: 0083-1786. ISBN: 978-0-16-000202-1. URL: https://personal.math.ubc.ca/~cbm/aands (besucht am 2025-09-05) (siehe S. 17–19, 151).

[2] Adam Aspin und Karine Aspin. *Query Answers with MariaDB – Volume I: Introduction to SQL Queries*. Tetras Publishing, Okt. 2018. ISBN: 978-1-9996172-4-0. See also[3] (siehe S. 140, 150).

[3] Adam Aspin und Karine Aspin. *Query Answers with MariaDB – Volume II: In-Depth Querying*. Tetras Publishing, Okt. 2018. ISBN: 978-1-9996172-5-7. See also[2] (siehe S. 140, 150).

[4] Daniel J. Barrett. *Efficient Linux at the Command Line*. Sebastopol, CA, USA: O'Reilly Media, Inc., Feb. 2022. ISBN: 978-1-0981-1340-7 (siehe S. 150, 152).

[5] Daniel Bartholomew. *Learning the MariaDB Ecosystem: Enterprise-level Features for Scalability and Availability*. New York, NY, USA: Apress Media, LLC, Okt. 2019. ISBN: 978-1-4842-5514-8 (siehe S. 150).

[6] Tim Berners-Lee. *Re: Qualifiers on Hypertext links...* Geneva, Switzerland: World Wide Web project, European Organization for Nuclear Research (CERN) und Newsgroups: alt.hypertext, 6. Aug. 1991. URL: https://www.w3.org/People/Berners-Lee/1991/08/art-6484.txt (besucht am 2025-02-05) (siehe S. 152).

[7] Alex Berson. *Client/Server Architecture*. 2. Aufl. Computer Communications Series. New York, NY, USA: McGraw-Hill, 29. März 1996. ISBN: 978-0-07-005664-0 (siehe S. 149).

[8] Silvia Botros und Jeremy Tinley. *High Performance MySQL*. 4. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Nov. 2021. ISBN: 978-1-4920-8051-0 (siehe S. 150).

[9] Ed Bott. *Windows 11 Inside Out*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., Feb. 2023. ISBN: 978-0-13-769132-6 (siehe S. 150).

[10] Ron Brash und Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2018. ISBN: 978-1-78862-936-2 (siehe S. 149).

References II

[11] Jason Cannon. *High Availability for the LAMP Stack*. Shelter Island, NY, USA: Manning Publications, Juni 2022 (siehe S. 150, 151).

[12] Donald D. Chamberlin. "50 Years of Queries". *Communications of the ACM (CACM)* 67(8):110–121, Aug. 2024. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/3649887. URL: https://cacm.acm.org/research/50-years-of-queries (besucht am 2025-01-09) (siehe S. 151).

[13] David Clinton und Christopher Negus. *Ubuntu Linux Bible*. 10. Aufl. Bible Series. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., 10. Nov. 2020. ISBN: 978-1-119-72233-5 (siehe S. 152).

[14] Edgar Frank „Ted" Codd. "A Relational Model of Data for Large Shared Data Banks". *Communications of the ACM (CACM)* 13(6):377–387, Juni 1970. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/362384.362685. URL: https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf (besucht am 2025-01-05) (siehe S. 151).

[15] *Database Language SQL*. Techn. Ber. ANSI X3.135-1986. Washington, D.C., USA: American National Standards Institute (ANSI), 1986 (siehe S. 151).

[16] Matt David und Blake Barnhill. *How to Teach People SQL*. San Francisco, CA, USA: The Data School, Chart.io, Inc., 10. Dez. 2019–10. Apr. 2023. URL: https://dataschool.com/how-to-teach-people-sql (besucht am 2025-02-27) (siehe S. 151).

[17] *Database Language SQL*. International Standard ISO 9075-1987. Geneva, Switzerland: International Organization for Standardization (ISO), 1987 (siehe S. 151).

[18] Paul Deitel, Harvey Deitel und Abbey Deitel. *Internet & World Wide WebW[: How to Program*. 5. Aufl. Hoboken, NJ, USA: Pearson Education, Inc., Nov. 2011. ISBN: 978-0-13-299045-5 (siehe S. 152).

[19] Justin Dennison, Cherokee Boose und Peter van Rysdam. *Intro to NumPy*. Centennial, CO, USA: ACI Learning. Birmingham, England, UK: Packt Publishing Ltd, Juni 2024. ISBN: 978-1-83620-863-1 (siehe S. 150).

[20] Russell J.T. Dyer. *Learning MySQL and MariaDB*. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2015. ISBN: 978-1-4493-6290-4 (siehe S. 150).

References III

[21] Leonhard Euler. "An Essay on Continued Fractions". Übers. von Myra F. Wyman und Bostwick F. Wyman. *Mathematical Systems Theory* 18(1):295–328, Dez. 1985. New York, NY, USA: Springer Science+Business Media, LLC. ISSN: 1432-4350. doi:10.1007/BF01699475. URL: https://www.researchgate.net/publication/301720080 (besucht am 2024-09-24). Translation of[22]. (Siehe S. 142).

[22] Leonhard Euler. "De Fractionibus Continuis Dissertation". *Commentarii Academiae Scientiarum Petropolitanae* 9:98–137, 1737–1744. Petropolis (St. Petersburg), Russia: Typis Academiae. URL: https://scholarlycommons.pacific.edu/cgi/viewcontent.cgi?article=1070 (besucht am 2024-09-24). See[21] for a translation. (Siehe S. 142, 153).

[23] Merran Evans, Nicholas Hastings und Brian Peacock. *Statistical Distributions*. 3. Aufl. Chichester, West Sussex, England, UK: Wiley Interscience, Juni 2000. ISBN: 978-0-471-37124-3 (siehe S. 17–19, 151).

[24] Luca Ferrari und Enrico Pirozzi. *Learn PostgreSQL*. 2. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Okt. 2023. ISBN: 978-1-83763-564-1 (siehe S. 151).

[25] Michael Filaseta. "The Transcendence of e and π". In: *Math 785: Transcendental Number Theory*. Columbia, SC, USA: University of South Carolina, Frühling 2011. Kap. 6. URL: https://people.math.sc.edu/filaseta/gradcourses/Math785/Math785Notes6.pdf (besucht am 2024-07-05) (siehe S. 152, 153).

[26] Michael Goodwin. *What is an API?* Armonk, NY, USA: International Business Machines Corporation (IBM), 9. Apr. 2024. URL: https://www.ibm.com/topics/api (besucht am 2024-12-12) (siehe S. 149).

[27] Terry Halpin und Tony Morgan. *Information Modeling and Relational Databases*. 3. Aufl. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Juli 2024. ISBN: 978-0-443-23791-1 (siehe S. 151).

[28] Jan L. Harrington. *Relational Database Design and Implementation*. 4. Aufl. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Apr. 2016. ISBN: 978-0-12-849902-3 (siehe S. 151).

[29] Charles R. Harris, K. Jarrod Millman, Stéfan van der Walt, Ralf Gommers, Pauli „pv" Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke und Travis E. Oliphant. "Array programming with NumPy". *Nature* 585:357–362, 2020. London, England, UK: Springer Nature Limited. ISSN: 0028-0836. doi:10.1038/S41586-020-2649-2 (siehe S. 150).

References IV

[30] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (siehe S. 150).

[31] Matthew Helmke. *Ubuntu Linux Unleashed 2021 Edition*. 14. Aufl. Reading, MA, USA: Addison-Wesley Professional, Aug. 2020. ISBN: 978-0-13-668539-5 (siehe S. 150, 152).

[32] Trey Hunner. "Mutable Default Arguments". In: *Python Morsels*. Reykjavík, Iceland: Python Morsels, 7. Apr. 2025. URL: https://www.pythonmorsels.com/mutable-default-arguments (besucht am 2025-09-06) (siehe S. 46–60).

[33] John Hunt. *A Beginners Guide to Python 3 Programming*. 2. Aufl. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2023. ISBN: 978-3-031-35121-1. doi:10.1007/978-3-031-35122-8 (siehe S. 151).

[34] John D. Hunter. "Matplotlib: A 2D Graphics Environment". *Computing in Science & Engineering* 9(3):90–95, Mai–Juni 2007. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 1521-9615. doi:10.1109/MCSE.2007.55 (siehe S. 150).

[35] John D. Hunter, Darren Dale, Eric Firing, Michael Droettboom und The Matplotlib Development Team. *Matplotlib: Visualization with Python*. Austin, TX, USA: NumFOCUS, Inc., 2012–2025. URL: https://matplotlib.org (besucht am 2025-02-02) (siehe S. 150).

[36] *Information Technology – Database Languages – SQL – Part 1: Framework (SQL/Framework), Part 1*. International Standard ISO/IEC 9075-1:2023(E), Sixth Edition, (ANSI X3.135). Geneva, Switzerland: International Organization for Standardization (ISO) und International Electrotechnical Commission (IEC), Juni 2023. URL: https://standards.iso.org/ittf/PubliclyAvailableStandards/ISO_IEC_9075-1_2023_ed_6_-_id_76583_Publication_PDF_(en).zip (besucht am 2025-01-08). Consists of several parts, see https://modern-sql.com/standard for information where to obtain them. (Siehe S. 151).

[37] Robert Johansson. *Numerical Python: Scientific Computing and Data Science Applications with NumPy, SciPy and Matplotlib*. New York, NY, USA: Apress Media, LLC, Dez. 2018. ISBN: 978-1-4842-4246-9 (siehe S. 150, 151).

[38] Arthur Jones, Kenneth R. Pearson und Sidney A. Morris. "Transcendence of e and π". In: *Abstract Algebra and Famous Impossibilities*. Universitext (UTX). New York, NY, USA: Springer New York, 1991. Kap. 9, S. 115–161. ISSN: 0172-5939. ISBN: 978-1-4419-8552-1. doi:10.1007/978-1-4419-8552-1_8 (siehe S. 152, 153).

References V

[39] Jay LaCroix. *Mastering Ubuntu Server*. 4. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2022. ISBN: 978-1-80323-424-3 (siehe S. 151).

[40] Łukasz Langa. *Literature Overview for Type Hints*. Python Enhancement Proposal (PEP) 482. Beaverton, OR, USA: Python Software Foundation (PSF), 8. Jan. 2015. URL: https://peps.python.org/pep-0482 (besucht am 2024-10-09) (siehe S. 152).

[41] Kent D. Lee und Steve Hubbard. *Data Structures and Algorithms with Python*. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2015. ISBN: 978-3-319-13071-2. doi:10.1007/978-3-319-13072-9 (siehe S. 151).

[42] Michael Lee, Ivan Levkivskyi und Jukka Lehtosalo. *Literal Types*. Python Enhancement Proposal (PEP) 586. Beaverton, OR, USA: Python Software Foundation (PSF), 14. März 2019. URL: https://peps.python.org/pep-0586 (besucht am 2024-12-17) (siehe S. 150).

[43] Jukka Lehtosalo, Ivan Levkivskyi, Jared Hance, Ethan Smith, Guido van Rossum, Jelle „JelleZijlstra“ Zijlstra, Michael J. Sullivan, Shantanu Jain, Xuanda Yang, Jingchen Ye, Nikita Sobolev und Mypy Contributors. *Mypy – Static Typing for Python*. San Francisco, CA, USA: GitHub Inc, 2024. URL: https://github.com/python/mypy (besucht am 2024-08-17) (siehe S. 150).

[44] Gloria Lotha, Aakanksha Gaur, Erik Gregersen, Swati Chopra und William L. Hosch. “Client-Server Architecture”. In: *Encyclopaedia Britannica*. Hrsg. von The Editors of Encyclopaedia Britannica. Chicago, IL, USA: Encyclopædia Britannica, Inc., 3. Jan. 2025. URL: https://www.britannica.com/technology/client-server-architecture (besucht am 2025-01-20) (siehe S. 149).

[45] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 151).

[46] *MariaDB Server Documentation*. Milpitas, CA, USA: MariaDB, 2025. URL: https://mariadb.com/kb/en/documentation (besucht am 2025-04-24) (siehe S. 150).

[47] “Mathematical Functions and Operators”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. 9.3. URL: https://www.postgresql.org/docs/17/functions-math.html (besucht am 2025-02-27) (siehe S. 152, 153).

[48] Jim Melton und Alan R. Simon. *SQL: 1999 – Understanding Relational Language Components*. The Morgan Kaufmann Series in Data Management Systems. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Juni 2001. ISBN: 978-1-55860-456-8 (siehe S. 151).

References VI

[49] Cameron Newham und Bill Rosenblatt. *Learning the Bash Shell – Unix Shell Programming: Covers Bash 3.0*. 3. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., 2005. ISBN: 978-0-596-00965-6 (siehe S. 149).

[50] Ivan Niven. "The Transcendence of π". *The American Mathematical Monthly* 46(8):469–471, Okt. 1939. London, England, UK: Taylor and Francis Ltd. ISSN: 1930-0972. doi:10.2307/2302515 (siehe S. 152).

[51] NumPy Team. *NumPy*. San Francisco, CA, USA: GitHub Inc und Austin, TX, USA: NumFOCUS, Inc. URL: https://numpy.org (besucht am 2025-02-02) (siehe S. 150).

[52] Regina O. Obe und Leo S. Hsu. *PostgreSQL: Up and Running*. 3. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Okt. 2017. ISBN: 978-1-4919-6336-4 (siehe S. 151).

[53] Robert Orfali, Dan Harkey und Jeri Edwards. *Client/Server Survival Guide*. 3. Aufl. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., 25. Jan. 1999. ISBN: 978-0-471-31615-2 (siehe S. 149).

[54] Ashwin Pajankar. *Hands-on Matplotlib: Learn Plotting and Visualizations with Python 3*. New York, NY, USA: Apress Media, LLC, Nov. 2021. ISBN: 978-1-4842-7410-1 (siehe S. 150).

[55] Yasset Pérez-Riverol, Laurent Gatto, Rui Wang, Timo Sachsenberg, Julian Uszkoreit, Felipe da Veiga Leprevost, Christian Fufezan, Tobias Ternent, Stephen J. Eglen, Daniel S. Katz, Tom J. Pollard, Alexander Konovalov, Robert M. Flight, Kai Blin und Juan Antonio Vizcaíno. "Ten Simple Rules for Taking Advantage of Git and GitHub". *PLOS Computational Biology* 12(7), 14. Juli 2016. San Francisco, CA, USA: Public Library of Science (PLOS). ISSN: 1553-7358. doi:10.1371/JOURNAL.PCBI.1004947 (siehe S. 149).

[56] *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 2025. URL: https://www.postgresql.org/docs/17/index.html (besucht am 2025-02-25).

[57] *PostgreSQL Essentials: Leveling Up Your Data Work*. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2024 (siehe S. 151).

[58] Abhishek Ratan, Eric Chou, Pradeeban Kathiravelu und Dr. M.O. Faruque Sarker. *Python Network Programming*. Birmingham, England, UK: Packt Publishing Ltd, Jan. 2019. ISBN: 978-1-78883-546-6 (siehe S. 149).

[59] Federico Razzoli. *Mastering MariaDB*. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2014. ISBN: 978-1-78398-154-0 (siehe S. 150).

References VII

[60] Mike Reichardt, Michael Gundall und Hans D. Schotten. "Benchmarking the Operation Times of NoSQL and MySQL Databases for Python Clients". In: *47th Annual Conference of the IEEE Industrial Electronics Society (IECON'2021*. 13.–15. Okt. 2021, Toronto, ON, Canada. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE), 2021, S. 1–8. ISSN: 2577-1647. ISBN: 978-1-6654-3554-3. doi:10.1109/IECON48115.2021.9589382 (siehe S. 150).

[61] Mark Richards und Neal Ford. *Fundamentals of Software Architecture: An Engineering Approach*. Sebastopol, CA, USA: O'Reilly Media, Inc., Jan. 2020. ISBN: 978-1-4920-4345-4 (siehe S. 149).

[62] Ellen Siever, Stephen Figgins, Robert Love und Arnold Robbins. *Linux in a Nutshell*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2009. ISBN: 978-0-596-15448-6 (siehe S. 150).

[63] Anna Skoulikari. *Learning Git*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mai 2023. ISBN: 978-1-0981-3391-7 (siehe S. 149).

[64] John Miles Smith und Philip Yen-Tang Chang. "Optimizing the Performance of a Relational Algebra Database Interface". *Communications of the ACM (CACM)* 18(10):568–579, Okt. 1975. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/361020.361025 (siehe S. 151).

[65] "SQL Commands". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. Part VI. Reference. URL: https://www.postgresql.org/docs/17/sql-commands.html (besucht am 2025-02-25) (siehe S. 151).

[66] Ryan K. Stephens und Ronald R. Plew. *Sams Teach Yourself SQL in 21 Days*. 4. Aufl. Sams Tech Yourself. Indianapolis, IN, USA: SAMS Technical Publishing und Hoboken, NJ, USA: Pearson Education, Inc., Okt. 2002. ISBN: 978-0-672-32451-2 (siehe S. 146, 151).

[67] Ryan K. Stephens, Ronald R. Plew, Bryan Morgan und Jeff Perkins. *SQL in 21 Tagen. Die Datenbank-Abfragesprache SQL vollständig erklärt (in 14/21 Tagen)*. 6. Aufl. Burgthann, Bayern, Germany: Markt+Technik Verlag GmbH, Feb. 1998. ISBN: 978-3-8272-2020-2. Translation of[66] (siehe S. 151).

[68] Allen Taylor. *Introducing SQL and Relational Databases*. New York, NY, USA: Apress Media, LLC, Sep. 2018. ISBN: 978-1-4842-3841-7 (siehe S. 151).

References VIII

[69] Alkin Tezuysal und Ibrar Ahmed. *Database Design and Modeling with PostgreSQL and MySQL*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2024. ISBN: 978-1-80323-347-5 (siehe S. 150, 151).

[70] ."Literals". In: *Static Typing with Python*. Hrsg. von The Python Typing Team. Beaverton, OR, USA: Python Software Foundation (PSF), 2021. URL: https://typing.python.org/en/latest/spec/literal.html (besucht am 2025-08-29) (siehe S. 150).

[71] Linus Torvalds. "The Linux Edge". *Communications of the ACM (CACM)* 42(4):38–39, Apr. 1999. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/299157.299165 (siehe S. 150).

[72] Mariot Tsitoara. *Beginning Git and GitHub: Version Control, Project Management and Teamwork for the New Developer*. New York, NY, USA: Apress Media, LLC, März 2024. ISBN: 979-8-8688-0215-7 (siehe S. 149, 152).

[73] Guido van Rossum und Łukasz Langa. *Type Hints*. Python Enhancement Proposal (PEP) 484. Beaverton, OR, USA: Python Software Foundation (PSF), 29. Sep. 2014. URL: https://peps.python.org/pep-0484 (besucht am 2024-08-22) (siehe S. 152).

[74] Sander van Vugt. *Linux Fundamentals*. 2. Aufl. Hoboken, NJ, USA: Pearson IT Certification, Juni 2022. ISBN: 978-0-13-792931-3 (siehe S. 150).

[75] Pauli „pv" Virtanen, Ralf Gommers, Travis E. Oliphant, Matt „mdhaber" Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, Ilhan „ilayn" Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregos, Paul van Mulbregt und SciPy 1.0 Contributors. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". *Nature Methods* 17:261–272, 2. März 2020. London, England, UK: Springer Nature Limited. ISSN: 1548-7091. doi:10.1038/s41592-019-0686-2. URL: http://arxiv.org/abs/1907.10121 (besucht am 2024-06-26). See also arXiv:1907.10121v1 [cs.MS] 23 Jul 2019. (Siehe S. 151).

[76] Christian Walck. *Hand-Book on Statistical Distributions for Experimentalists*. Internal Report SUF-PFY/96-01. Stockholm, Sweden: University of Stockholm, 11. Dez. 1996–10. Sep. 2007. URL: https://www.stat.rice.edu/~dobelman/textfiles/DistributionsHandbook.pdf (besucht am 2025-09-05) (siehe S. 17–19, 151).

References IX

[77] Thomas Weise (汤卫思). *Databases*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), Institute of Applied Optimization (应用优化研究所, IAO), 2025. URL: https://thomasweise.github.io/databases (besucht am 2025-01-05) (siehe S. 149, 151).

[78] Thomas Weise (汤卫思). *Programming with Python*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), Institute of Applied Optimization (应用优化研究所, IAO), 2024–2025. URL: https://thomasweise.github.io/programmingWithPython (besucht am 2025-01-05) (siehe S. 150, 151).

[79] *What is a Relational Database?* Armonk, NY, USA: International Business Machines Corporation (IBM), 20. Okt. 2021–12. Dez. 2024. URL: https://www.ibm.com/think/topics/relational-databases (besucht am 2025-01-05) (siehe S. 151).

[80] Ulf Michael „Monty“ Widenius, David Axmark und Uppsala, Sweden: MySQL AB. *MySQL Reference Manual – Documentation from the Source*. Sebastopol, CA, USA: O'Reilly Media, Inc., 9. Juli 2002. ISBN: 978-0-596-00265-7 (siehe S. 150).

[81] Kinza Yasar und Craig S. Mullins. *Definition: Database Management System (DBMS)*. Newton, MA, USA: TechTarget, Inc., Juni 2024. URL: https://www.techtarget.com/searchdatamanagement/definition/database-management-system (besucht am 2025-01-11) (siehe S. 149).

[82] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, Juni 2017. ISBN: 978-1-78439-687-9 (siehe S. 149).

Glossary (in English) I

API An *Application Programming Interface* is a set of rules or protocols that enables one software application or component to use or communicate with another[26].

Bash is a the shell used under Ubuntu Linux, i.e., the program that „runs“ in the terminal and interprets your commands, allowing you to start and interact with other programs[10,49,82]. Learn more at https://www.gnu.org/software/bash.

client In a client-server architecture, the client is a device or process that requests a service from the server. It initiates the communication with the server, sends a request, and receives the response with the result of the request. Typical examples for clients are web browsers in the internet as well as clients for database management systems (DBMSes), such as psql.

client-server architecture is a system design where a central server receives requests from one or multiple clients[7,44,53,58,61]. These requests and responses are usually sent over network connections. A typical example for such a system is the World Wide Web (WWW), where web servers host websites and make them available to web browsers, the clients. Another typical example is the structure of database (DB) software, where a central server, the DBMS, offers access to the DB to the different clients. Here, the client can be some terminal software shipping with the DBMS, such as psql, or the different applications that access the DBs.

DB A *database* is an organized collection of structured information or data, typically stored electronically in a computer system. Databases are discussed in our book *Databases*[77].

DBMS A *database management system* is the software layer located between the user or application and the DB. The DBMS allows the user/application to create, read, write, update, delete, and otherwise manipulate the data in the DB[81].

Git is a distributed Version Control Systems (VCS) which allows multiple users to work on the same code while preserving the history of the code changes[63,72]. Learn more at https://git-scm.com.

GitHub is a website where software projects can be hosted and managed via the Git VCS[55,72]. Learn more at https://github.com.

IT information technology

Glossary (in English) II

LAMP Stack A system setup for web applications: Linux, Apache (a web server), MySQL, and the server-side scripting language PHP[11,31].

Linux is the leading open source operating system, i.e., a free alternative for Microsoft Windows[4,30,62,71,74]. We recommend using it for this course, for software development, and for research. Learn more at https://www.linux.org. Its variant Ubuntu is particularly easy to use and install.

literal A literal is a specific concrete value, something that is written down as-is[42,70]. In Python, for example, `"abc"` is a string literal, `5` is an integer literal, and `23.3` is a `float` literal. In contrast, `sin(3)` is not a literal. Also, while `5` is an integer literal, if we create a variable `a = 5` then `a` is not a literal either (it is a variable). Hence, literals are values that the Python interpreter reads directly from the source code and creates as objects in memory. They are not something that is the result from a computation or the result of a variable lookup. Python supports some type hints for literals, including the type `LiteralString` for string literals and the type `Literal[xyz]` for arbitrary literals `xyz`.

MariaDB An open source relational database management system that has forked off from MySQL[2,3,5,20,46,59]. See https://mariadb.org for more information.

Matplotlib is a Python package for plotting diagrams and charts[34,35,37,54]. Learn more at at https://matplotlib.org[35].

Microsoft Windows is a commercial proprietary operating system[9]. It is widely spread, but we recommend using a Linux variant such as Ubuntu for software development and for our course. Learn more at https://www.microsoft.com/windows.

Mypy is a static type checking tool for Python[43] that makes use of type hints. Learn more at https://github.com/python/mypy and in[78].

MySQL An open source relational database management system[8,20,60,69,80]. MySQL is famous for its use in the LAMP Stack. See https://www.mysql.com for more information.

NumPy is a fundamental package for scientific computing with Python, which offers efficient array datastructures[19,29,37]. Learn more at https://numpy.org[51].

Glossary (in English) III

PDF *Probability Density Function*[1,23,76] of a continuous random variable $\mathcal{X}$ that can take on values from range $\Omega \subseteq \mathbb{R}$ is a function $f_{\mathcal{X}} : \Omega \mapsto \mathbb{R}^+$ such that

- $f_{\mathcal{X}}(x) > 0$ for all $x \in \Omega$ (non-negativity),
- $\int_{\Omega} f_{\mathcal{X}}(x)dx = 1$ (normalization), and
- the probability that $x \in A$ for all $A \subseteq \Omega$ is $\int_A f_{\mathcal{X}}(x)dx$.

PostgreSQL An open source object-relational DBMS[24,52,57,69]. See https://postgresql.org for more information.

psql is the client program used to access the PostgreSQL DBMS server.

Python The Python programming language[33,41,45,78], i.e., what you will learn about in our book[78]. Learn more at https://python.org.

relational database A relational DB is a database that organizes data into rows (tuples, records) and columns (attributes), which collectively form tables (relations) where the data points are related to each other[14,27,28,64,68,77,79].

SciPy is a Python library for scientific computing[37,75]. Learn more at https://scipy.org.

server In a client-server architecture, the server is a process that fulfills the requests of the clients. It usually waits for incoming communication carring the requests from the clients. For each request, it takes the necessary actions, performs the required computations, and then sends a response with the result of the request. Typical examples for servers are web servers[11] in the internet as well as DBMSes. It is also common to refer to the computer running the server processes as server as well, i.e., to call it the „server computer“[39].

SQL The *Structured Query Language* is basically a programming language for querying and manipulating relational databases[12,15–17,36,48,65–68]. It is understood by many DBMSes. You find the Structured Query Language (SQL) commands supported by PostgreSQL in the reference[65].

Glossary (in English) IV

terminal
: A terminal is a text-based window where you can enter commands and execute them[4,13]. Knowing what a terminal is and how to use it is very essential in any programming- or system administration-related task. If you want to open a terminal under Microsoft Windows, you can Druck auf ⊞+R, dann Schreiben von `cmd`, dann Druck auf ↵. Under Ubuntu Linux, Ctrl+Alt+T opens a terminal, which then runs a Bash shell inside.

type hint
: are annotations that help programmers and static code analysis tools such as Mypy to better understand what type a variable or function parameter is supposed to be[40,73]. Python is a dynamically typed programming language where you do not need to specify the type of, e.g., a variable. This creates problems for code analysis, both automated as well as manual: For example, it may not always be clear whether a variable or function parameter should be an integer or floating point number. The annotations allow us to explicitly state which type is expected. They are *ignored* during the program execution. They are a basically a piece of documentation.

Ubuntu
: is a variant of the open source operating system Linux[13,31]. We recommend that you use this operating system to follow this class, for software development, and for research. Learn more at https://ubuntu.com. If you are in China, you can download it from https://mirrors.ustc.edu.cn/ubuntu-releases.

VCS
: A *Version Control System* is a software which allows you to manage and preserve the historical development of your program code[72]. A distributed VCS allows multiple users to work on the same code and upload their changes to the server, which then preserves the change history. The most popular distributed VCS is Git.

WWW
: World Wide Web[6,18]

π
: is the ratio of the circumference U of a circle and its diameter d, i.e., $\pi = U/d$. $\pi \in \mathbb{R}$ is an irrational and transcendental number[25,38,50], which is approximately $\pi \approx 3.141\,592\,653\,589\,793\,238\,462\,643$. In Python, it is provided by the `math` module as constant `pi` with value `3.141592653589793`. In PostgreSQL, it is provided by the SQL function `pi()` with value `3.141592653589793`[47].

Glossary (in English) V

e is Euler's number[22], the base of the natural logarithm. $e \in \mathbb{R}$ is an irrational and transcendental number[25,38], which is approximately $e \approx 2.718\,281\,828\,459\,045\,235\,360$. In Python, it is provided by the `math` module as constant `e` with value `2.718281828459045`. In PostgreSQL, you can obtain it via the SQL function `exp(1)` as value `2.718281828459045`[47].

$\mathbb{R}$ the set of the real numbers.

$\mathbb{R}^+$ the set of the positive real numbers, i.e., $\mathbb{R}^+ = \{x \in \mathbb{R} : x > 0\}$.