Python Packages

Bas Terwijn

February 2, 2026

1 Introductie

Dit document behandelt het gebruik van Python packages. Om bijvoorbeeld package 'matplotlib' te gebruiken moeten we deze eerste importen met:

```
import matplotlib
```

Dit soort `import` statements schrijven we meestal bovenaan een Python file.

1.1 Doelen

Bekend raken met:

- package 'matplotlib'
- package 'math'
- package 'random'

2 Package matplotlib

Om data beter te begrijpen is het vaak nuttig om deze te visualiseren. Dit kan bv met package matplotlib, zie de Quick start guide. De visualisatie van function $y = x^2 + 3x - 5$ in figuur 1:

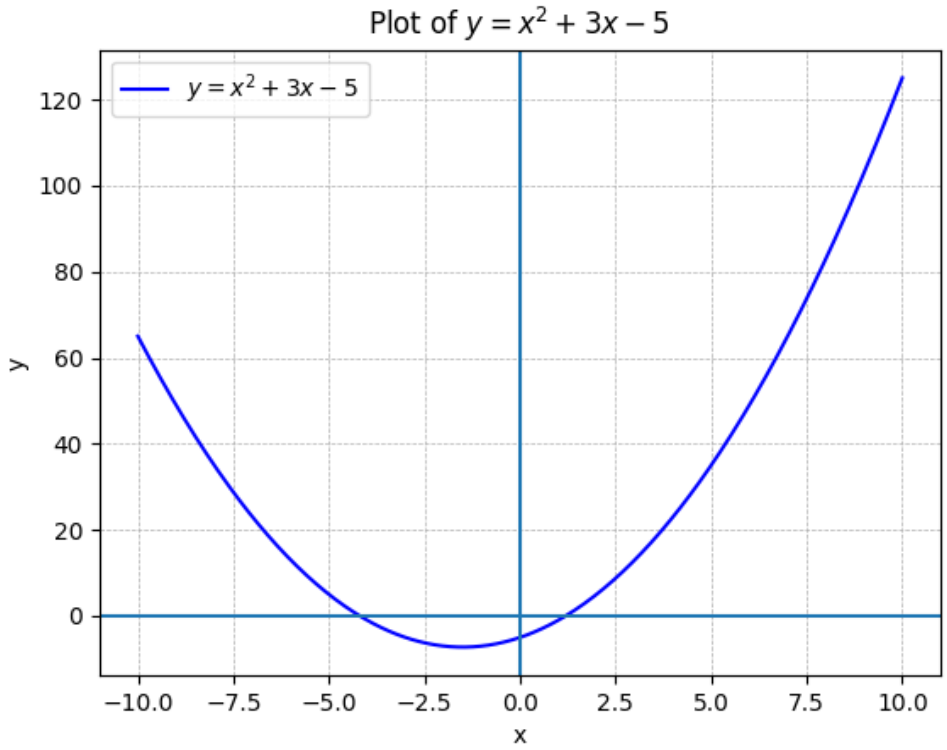

Figure 1: Matplotlib voorbeeld van parabool

is het resultaat van programma:

src/plot_parabola.py

```python
import matplotlib.pyplot as plt
import numpy as np

# Define the function
def f(x):
    return x**2 + 3*x - 5

# Generate x and y values
x = np.linspace(-10, 10, 400)  # x values from -10 to 10 with 400 points
y = f(x)                       # compute y value for each x value

# Plot the function
plt.plot(x, y, label=r"$y = x^2 + 3x - 5$", color='blue')

# Annotate the plot: add axes, grid, labels, legend, and title
plt.xlabel("x") # x-axis name
plt.ylabel("y") # y-axis name
plt.axhline(0)  # x-axis horizontal line
plt.axvline(0)  # y-axis vertical line
plt.grid(True, linestyle='--', linewidth=0.5) # show grid
plt.legend()                                  # add legend
plt.title("Plot of $y = x^2 + 3x - 5$")       # add title

# Show the plot
plt.show()
```

waarin we met package `numpy` een vector $\boldsymbol{x}$ aanmaken en voor elke x_i waarde in deze vector met functie $f(x)$ de

y waarde berekenen en plotten. Daarna volgt nog wat annotatie van de plot om deze duidelijker te maken.

2.1 Opdracht `approximate.py`

Benader de x-coordinaten van de snijpunten van functies:

$$f_1(x) = -0.0001242x^5 + 0.00181x^4 - 0.006097x^3 - 0.3235x^2 - 0.1989x + 2.488$$
$$f_2(x) = 0.00007053x^6 + 0.00003856x^5 - 0.01555x^4 - 0.02159x^3 + 0.7697x^2 + 1.206x - 5.734$$

in bestand `assignments/approximate.py`. Gebruik matplotlib voor een visualisatie om een eerste indruk te krijgen van hoeveel snijpunten er zijn en waar deze ongeveer liggen. Schrijf daarna Python code om het x-coordinaat van elk snijpunt zo dicht mogelijk te benaderen.

3 Package math

De math package geeft ons veel verschillende wiskundige functies.

3.1 Afronden

Daaronder vinden we ook enkele functies voor het afronden van een float waarde x naar een int (geheel getal).

- `math.trunc(x)` rond x af richting 0
- `math.floor(x)` rond x naar beneden af
- `math.ceil(x)` rond x naar boven af
- `round(x)` rond x af naar dichtsbijgelegen gehele getal

De expressie `int(x)` geeft hetzelfde resultaat als `math.trunc(x)`. In dit programma zien we deze functies toegepast op float 2.7.

src/round.py

```
import math

print( int(2.7)        )  # 2
print( math.trunc(2.7) )  # 2
print( math.floor(2.7) )  # 2
print( math.ceil(2.7)  )  # 3
print( round(2.7)      )  # 3
```

De verschillen tussen de afrondingsfuncties voor verschillende waarde van x zijn weergeven in figuur 2:

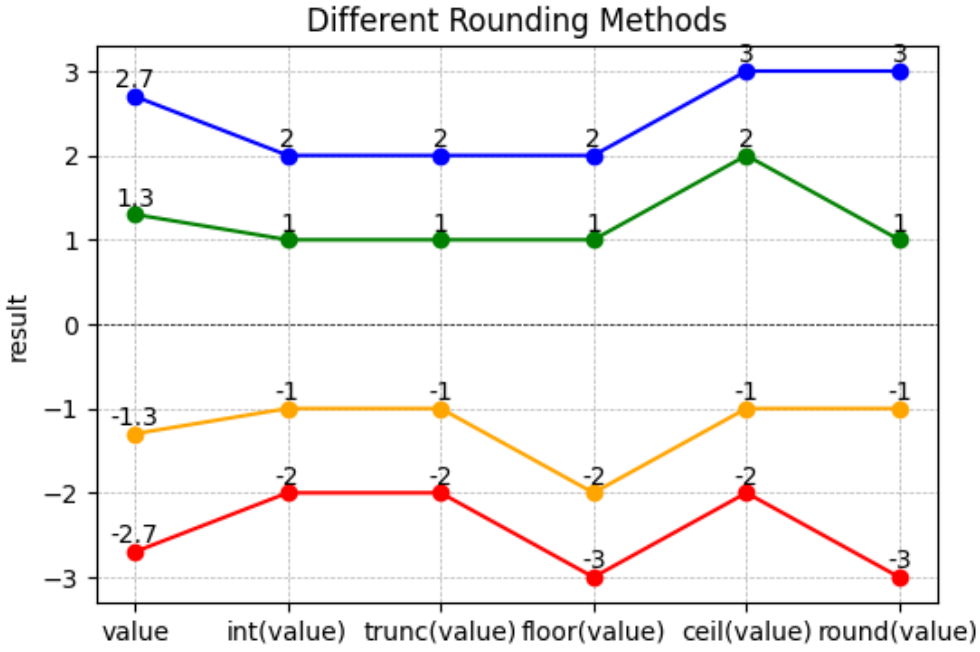

Figure 2: Visualisatie van verschillende afrondingsfuncties

Een eigenaardigheid van de `round(x)` functie is dat als x precies tussen twee gehele getallen in ligt (en dus een .5 waarde heeft) deze x dan wordt afgerond naar het dichtsbijzijnde even getal zoals weergeven in figuur 3. Dit wordt "banker's round" genoemd en zorgt dat veel verschillende floats met een .5 waarde ongeveer even vaak naar beneden als naar boven worden afgerond zodat een bankier geen grote positieve of negatieve effecten ziet bij het afronden van grote aantallen transacties naar hele centen.

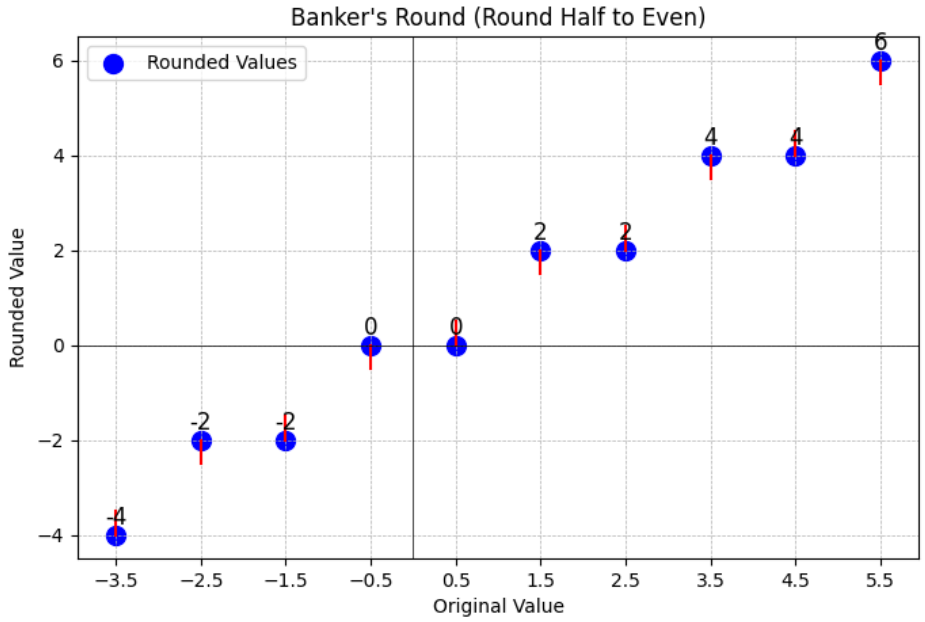

Figure 3: Banker's Round

Bij het berekenen van bijvoorbeeld tentamencijfers worden dit soort waarden liever altijd naar boven afgerond, een

5.5 wordt een 6 en een 6.5 een 7. Dit wordt ook wel “round half up” genoemd. Python biedt hier geen functie voor, maar we kunnen deze wel zelf definieeren als:

```
def round_half_up(x):
    return math.floor(x + 0.5)
```

wat de afronding geeft zoals in figuur 4 is weergegeven.

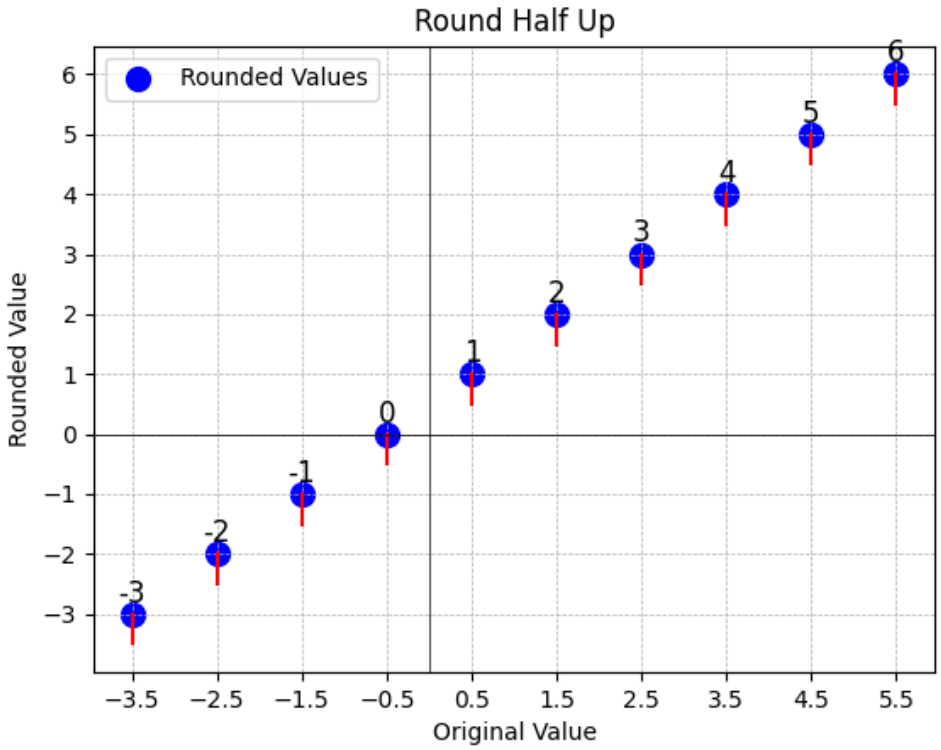

Figure 4: Round Half Up

Dit figuur is gemaakt met met deze Python code:

src/plot_round_half_up.py

```
import matplotlib.pyplot as plt
import math

def round_half_up(x):
    return math.floor(x + 0.5)

values = [ i/10 for i in range(-35, 60, 10)]
rounded_values = [round_half_up(x) for x in values]

# Create the plot
plt.scatter(values, rounded_values, color='blue', s=100, label="Rounded Values")

# Draw lines connecting original values to rounded values
for i in range(len(values)):
    plt.plot([values[i], values[i]], [values[i], rounded_values[i]],
             'r')  # Dashed red line
    plt.text(values[i], rounded_values[i] + 0.2, str(rounded_values[i]),
             ha='center', fontsize=12, color='black')
```

```
# Annotate graph
plt.axhline(0, color='black', linewidth=0.5)  # x-axis
plt.axvline(0, color='black', linewidth=0.5)  # y-axis
plt.grid(True, linestyle='--', linewidth=0.5)
plt.title("Round Half Up")
plt.xlabel("Original Value")
plt.ylabel("Rounded Value")
plt.xticks(values)  # Show exact test values on x-axis
plt.yticks(sorted(set(rounded_values)))  # Show only the rounded numbers
plt.legend()
plt.show()
```

3.2 Wiskundige functies

De faculteit (of in het Engels de factorial) van x, in Python geschreven als `math.factorial(x)`, wordt in de wiskunde ook wel geschreven als $x!$ en zijn alle waarden van 1 t/m x vermenigvuldigd. Dus bijvoorbeeld `math.factorial(5)` $= 5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$. Deze snelgroeiende functie is alleen gedefinieerd voor gehele niet-negatieve getallen, en 0! is gedefinieerd als 1, zie figuur 5.

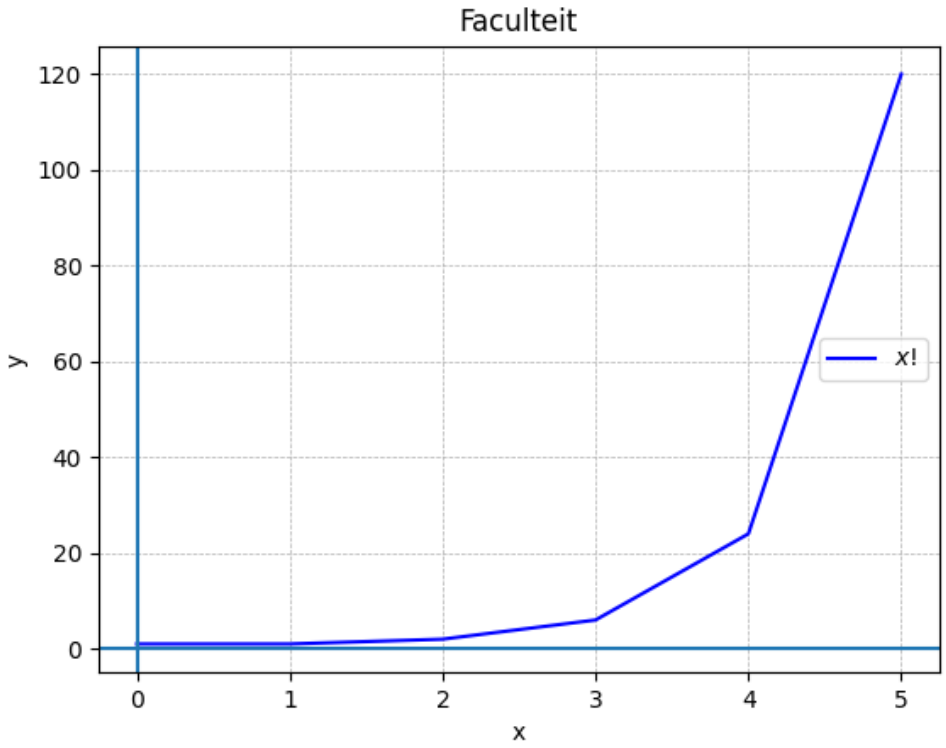

Figure 5: functie math.factorial(x)

De functie voor machtsverheffen is `math.pow(a, b)` wat de waarde a^b geeft. Dus bijvoorbeeld `math.pow(2, 3)` $= 2.0^{3.0} = 2.0 \cdot 2.0 \cdot 2.0 = 8.0$. Het verschil met `2**3` (wat we ook kunnen schrijven als `pow(2, 3)`, dus zonder `math.` ervoor) is dat `math.pow(2,3)` deze berekening altijd met floats uitvoert terwijl `2**3` dit met ints doet wanneer alle argumenten van het type int zijn. Omdat een int geen maximale waarde heeft zal `2**2000` resulteren in een exacte en hele grote int waarde:

```
114813069527425452423283320117768198402231770208869520047764273682576626139237031385665948631650626991844596463898746277344711896086305533142593135616665318539129989145312280000688779148240044871428926990063486244781615463646388363947317026040466353970904996558162398808944629605623311649536
```

```
164221970332681344168908984458505602379484807914058900934776500429002716706625830522008132236281291761267883317206598995396418127021779858404042159853183251540889433902091920554957783589672039160081957216630582755380425583726015528348786419432054508915275783882625175435528800822842770817965453762184851149029376
```

maar rekenen met dit soort grote getallen kan erg langzaam zijn. Floats hebben wel een maximale waarde en geven een benadering van de waarde, maar blijven daardoor wel snel om mee te rekenen. Welke vorm van machtsverheffen beter is, hangt van de situatie af. De functie `math.pow(2, x)` is weergegeven in figuur 6.

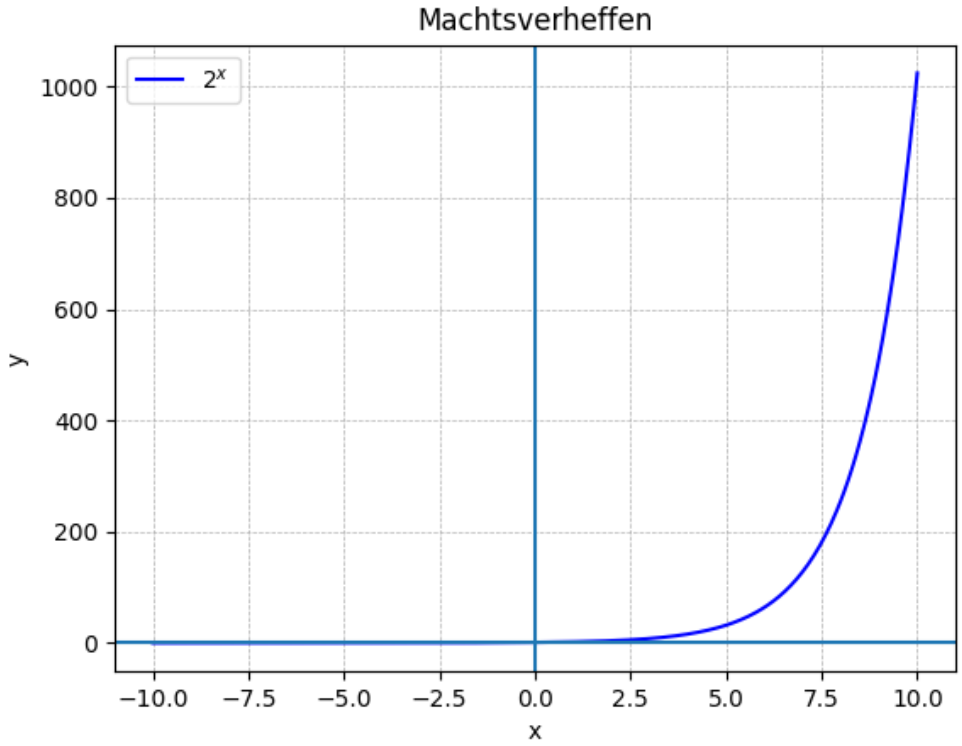

Figure 6: functie math.pow(2, x)

De machtsverheffen-functie heeft twee inverse-functies, één daarvan is zichzelf:

$$a^b = c \qquad \Leftrightarrow \qquad c^{\frac{1}{b}} = a$$

de andere is de logaritme functie die we schrijven als log():

$$a^b = c \qquad \Leftrightarrow \qquad \log_a(c) = b$$

Een voorbeeld, waar we duidelijk zien dat we met twee input waarden steeds de derde waarde kunnen berekenen, is:

$$2^{10} = 1024$$
$$1024^{\frac{1}{10}} = 2$$
$$\log_2(1024) = 10$$

De $\log_2(1024)$ schrijven we in Python als `math.log(1024, 2)`, dus het grondtal komt als tweede argument. Als we geen grondtal geven dan is de default waarde van het grondtal de constante e, wat we in Python schrijven

als `math.e`, en welke gelijk is aan 2.718281828459045. De functie $\log_e(x)$ schrijven we ook wel als $\ln(x)$ en wordt dan de natuurlijke logaritme functie genoemd. Omdat de `math.pow(2, x)` en `math.log(x, 2)` elkaars inverse zijn, kunnen we deze functies spiegelen met de lijn $y = x$ die we in rood weergeven in figuur 7. Dat is een goede manier om je de log() functie voor te stellen en maakt deze makkelijker te onthouden. Het laat ook zien waarom er geen negatieve waarden in een log() functie kunnen worden ingevoerd.

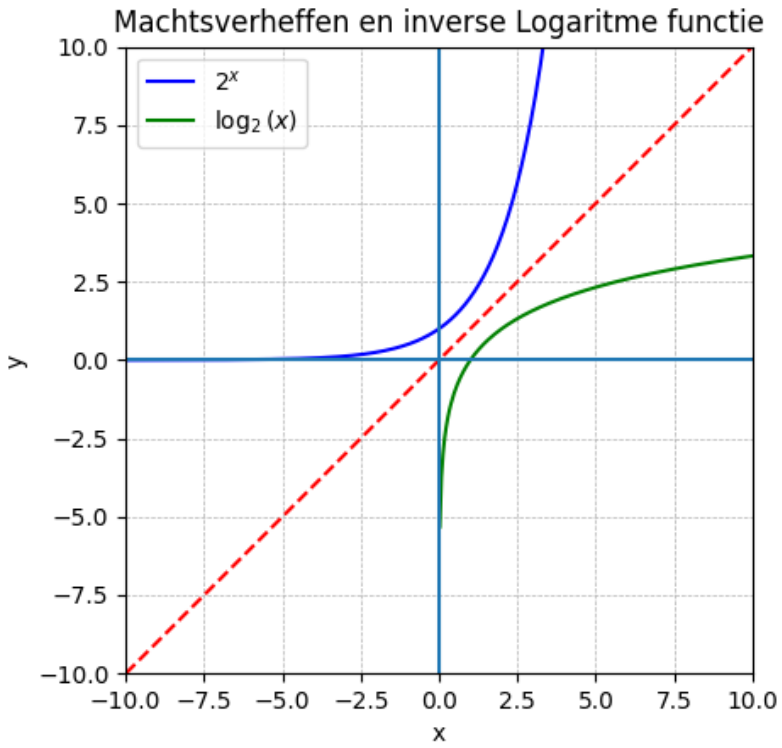

Figure 7: functie math.pow(2, x) en de inverse functie math.log(x, 2)

Dit figuur is gemaakt met deze code:

src/plot_pow_log.py

```python
import matplotlib.pyplot as plt
import numpy as np
import math

# Generate x and y values
domain = 10
all_x = np.linspace(-domain, domain, 400) # x values from -10 to 10 with 400 points
pow = [math.pow(2, x) for x in all_x]     # compute y value for each x value
pos_x = [x for x in all_x if x>0]     # only non-negative x
log = [math.log(x, 2) for x in pos_x] # compute y value for each x value

# Plot the functions
plt.plot(all_x, pow, label=r"$2^x$", color='blue')
plt.plot(pos_x, log, label=r"$\log_2(x)$", color='green')
plt.plot([-domain, domain], [-domain, domain], 'r--') # mirror line y=x

# Annotate the plot: add axes, grid, labels, legend, and title
plt.xlim(-domain, domain) # limit the x-axis
plt.ylim(-domain, domain) # limit the y-axis
plt.gca().set_aspect('equal', adjustable='box') # use equal x and y scale
```

```
plt.xlabel("x") # x-axis name
plt.ylabel("y") # y-axis name
plt.axhline(0)  # x-axis horizontal line
plt.axvline(0)  # y-axis vertical line
plt.grid(True, linestyle='--', linewidth=0.5) # show grid
plt.legend()                                  # add legend
plt.title("Machtsverheffen en inverse Logaritme functie") # add title

# Show the plot
plt.show()
```

De math package heeft nog veel andere wiskunde functies zoals `sin()`, `cos()`, `tan()` en hun inversen, maar die behandelen we hier niet.

3.3 Opdracht `combinatorics.py`

De faculteit, machtsverheffen en logaritme functies komen op veel verschillende plekken terug in uw opleiding. Hier gebruiken we ze bij het berekenen van combinatoriek, combinaties en permutaties. Lees hiervoor eerst de Combinations and Permutations pagina van de mathsisfun website, en beantwoord de vragen in `assignments/combinatorics.py`.

4 Package random

De random package bevat veel nuttige functies om pseudo-random waarden te genererenn. Met pseudo-random wordt bedoeld dat de waarden niet echt random zijn maar uit een deterministisch algorithme voortkomen, wat zorgt dat de waarden wel random lijken. De belangrijkste functies zijn:

src/random_examples.py

```
import random
random.seed(0)  # use same random numbers each run

a, b = 0, 4
print( random.randint(a, b)   ) # 3         random int from a through b
print( random.randrange(a, b) ) # 3         random int from a to b
print( random.random()        ) # 0.041...  random float from 0.0 to 1.0
print( random.uniform(a, b)   ) # 3.861...  random float from a to b

mylist = [1, 2, 3]
print( random.choice(mylist) )       # 2       choose 1 random value from list
print( random.choices(mylist, k=2) ) # [2, 3]  choose k values with repetition
print( random.sample(mylist, k=2) )  # [2, 3]  choose k values without repetition
print( random.shuffle(mylist) )      # None    gives mylist a random order

print( random.gauss(mu=0.0, sigma=1.0)) # -0.813... random float from
                                        #           normal distribution
```

De `random.seed(0)` functie-aanroep wordt hier gebruikt om te zorgen dat bij iedere run van deze Python file dezelfde pseudo-random waarden worden gegenereerd. Deze random waarden komen op een bepaalde manier voort uit een deterministisch algoritme met het argument '0' (de seed waarde) als enige input. Bij het testen van software kan het heel nuttig zijn om steeds dezelfde random waarden te genereren om bijoorbeeld een fout te kunnen reproduceren. Bij het aanroepen van deze functie met een andere constante, bv `random.seed(1)`, krijgen we bij iedere run ook steeds dezelfde random waarden, maar nu andere waarden dan bij een seed van 0. Als de functie `random.seed()` niet wordt aangeroepen dan wordt de huidige tijd als een seed gebruikt en worden er dus bij iedere run wel steeds andere random waarden gegenereerd.

Als je een seed gebruikt, dan wil de `random.seed(<constant>)` functie meestal maar één keer aan het begin van een programma aanroepen, omdat je anders, zoals in dit bijvoorbeeld, steeds dezelfde random waarde krijgt bij de eerst volgende aanroep van een random functie:

src/seed.py

```python
import random

for i in range(10):
    random.seed(0)  # resets the random seed in each iteration of the loop, BAD!
    print(random.randrange(10), end=' ')  # 6 6 6 6 6 6 6 6 6 6
```

Zo kunnen we bijvoorbeeld 50 random punten genereren met x en y waarden met gelijke (uniform) kans verdeeld over het interval -5 tot $+5$ en deze plotten met matplotlib in figuur 10

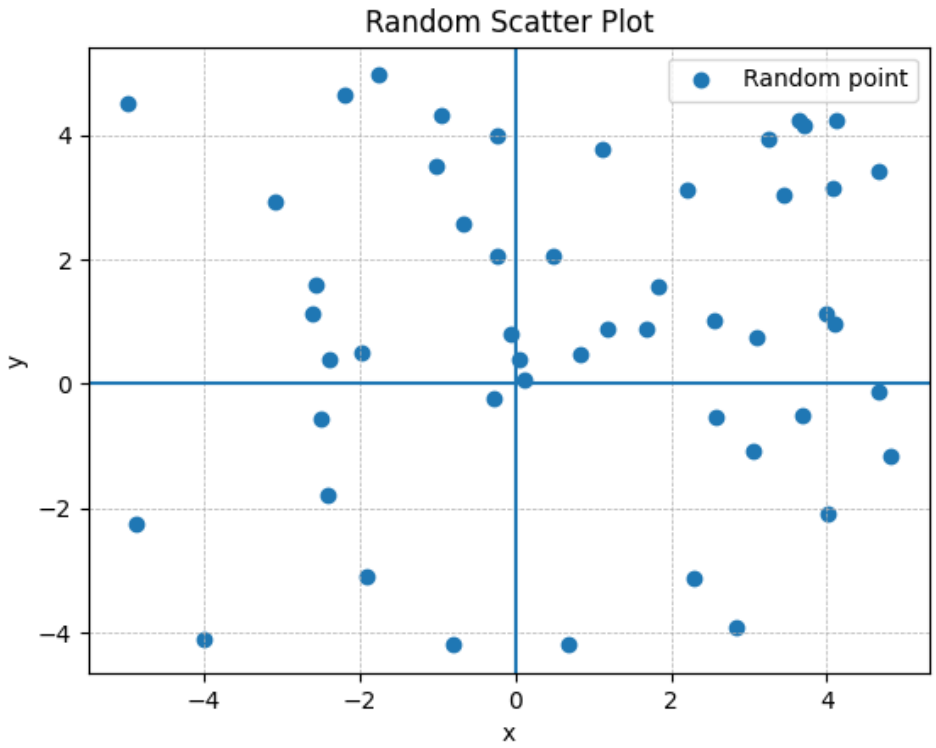

Figure 8: Matplotlib voorbeeld van scatter plot

met gebruik van deze Python code:

src/plot_scatter.py

```python
import matplotlib.pyplot as plt
import random
random.seed(0)  # same random numbers each run

# Generate random data points
x_values = [random.uniform(-5, 5) for _ in range(50)]  # 50 random x values
y_values = [random.uniform(-5, 5) for _ in range(50)]  # 50 random y values

# Create scatter plot
plt.scatter(x_values, y_values, label="Random point")

# Annotate the graph
plt.xlabel("x") # x-axis name
plt.ylabel("y") # y-axis name
plt.axhline(0)  # x-axis horizontal line
plt.axvline(0)  # y-axis vertical line
plt.grid(True, linestyle='--', linewidth=0.5)
plt.legend()
plt.title("Random Scatter Plot")

# Show the scatter plot
plt.show()
```

5 Normale of Gauss-verdeling

Naast random waarden met een gelijk (uniform) verdeelde kans tussen bepaalde waarden, kunnen we ook random waarden kiezen uit bijvoorbeeld de Normale of Gauss-verdeling. Hier hebben waarden die dichter bij μ ('mu', ook wel 'gemiddelde' of 'mean' genoemd) liggen een hogere kans om gekozen te worden. De σ ('sigma', ook wel 'standaardafwijking' of 'standard deviation' genoemd) waarde bepaalt hoe snel die kans afneemt naarmate een waarde verder van μ af ligt. De kans dat een random waarde tussen de -3σ en $+3\sigma$ van μ af ligt is 99.73% zoals weergegeven in figuur 9.

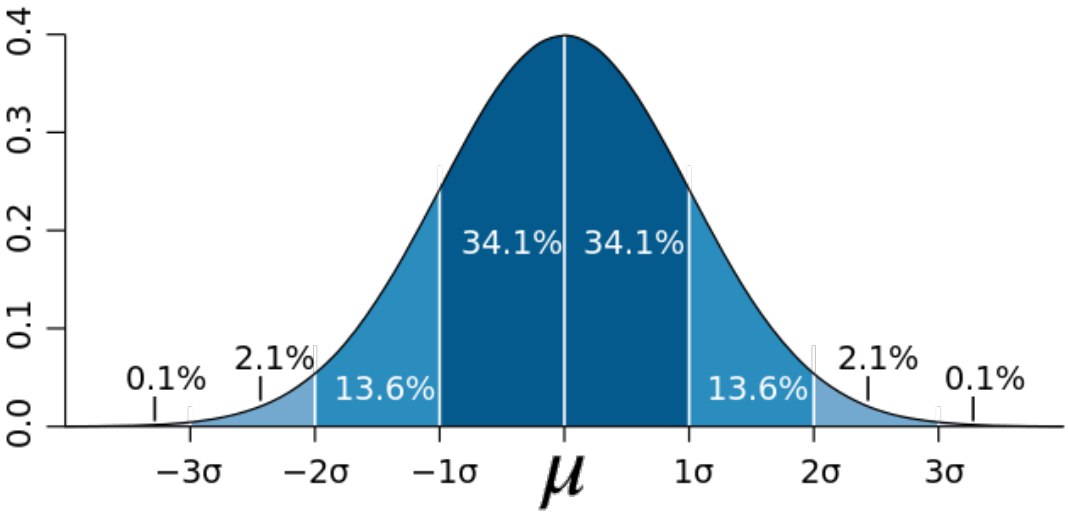

Figure 9: Normale of Gauss-verdeling

Zo kunnen we bijvoorbeeld 1000 random x waarden genereren aan de hand van een Gauss-verdeling met bijvoorbeeld $\mu = 0$ en $\sigma = 1$ en met matplotlib door middel van een histogram een telling weergeven van waar deze waarden dan gegenereerd worden, zoals in figuur 10

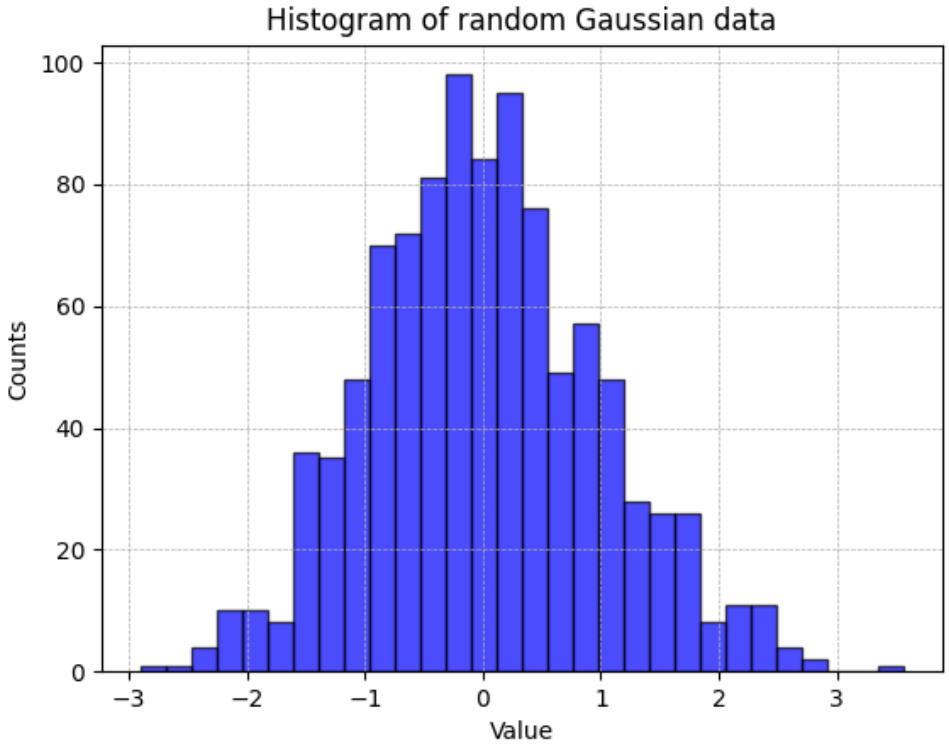

Figure 10: Matplotlib voorbeeld van een histogram

met gebruik van deze Python code:

src/plot_histogram.py

```python
import matplotlib.pyplot as plt
import random
random.seed(0)  # same random numbers each run

# Generate 1000 random values from a Gaussian distribution
data = [random.gauss(mu=0, sigma=1) for _ in range(1000)]

# Create histogram
plt.hist(data, bins=30, color='blue', edgecolor='black', alpha=0.7)

# Annotate the graph
plt.xlabel("Value")
plt.ylabel("Counts")
plt.grid(True, linestyle='--', linewidth=0.5)
plt.title("Histogram of random Gaussian data")

# Show the histogram
plt.show()
```

Hoe meer punten we genereren hoe meer dit histogram de vorm van de Gauss-verdeling aanneemt. Deze Gauss-verdeling komt in heel veel uiteenlopende problemen terug.