Inhaltsverzeichnis

1. Vorwort

In der letzten Zeit ist nichts Neues auf die Homepage gekommen, das soll aber nicht heißen, dass mein Neffe und ich untätig waren. Wir haben mit `pygame` weitere Spiele entwickelt, allerdings fehlte mir die Zeit, eine schöne Projektdokumentation dazu zu machen.

Jetzt haben wir uns entschieden, wieder mehr mit der Entwicklung von kleineren Modulen zu beschäftigen, was uns beiden einen neuen Horizont bieten soll.

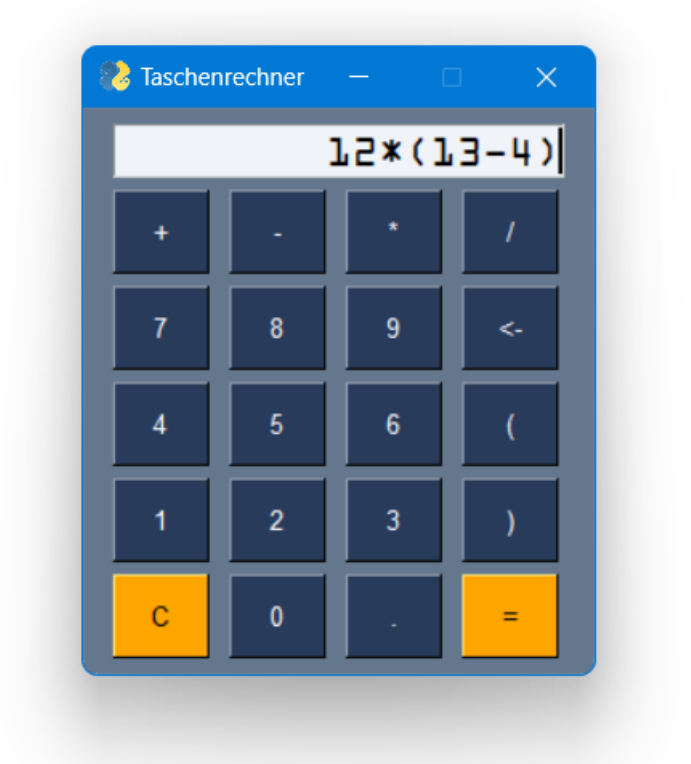
Was gibt es also diesmal zu entdecken? Wir werden einen Taschenrechner programmieren, wie er schon vielfach entwickelt und dokumentiert wurde. Das sollte uns genügend Zeit geben, auch noch einmal über grundsätzliche Python-Syntax und -Funktionen zu sprechen, sodass auch wieder Anfänger in der Lage sein sollten, ohne Probleme zu folgen.

Wie immer an dieser Stelle der Aufruf, meldet Euch gerne bei mir unter papa@papa-programmiert.de, ich bin für Kritik und Anregungen offen und immer zu haben.

Und jetzt viel Spaß!

2. Das Projekt

Wie im Vorwort beschrieben, geht es in diesem Projekt um einen Taschenrechner. Das Bild dazu sieht dann etwa so aus:



Der Code besteht sozusagen aus zwei Teilen, zum einen der Programmierung des Rechnens selbst und zum anderen dann auch die graphische Oberfläche oder „GUI“.

Für die GUI haben wir uns für die Bibliothek `PySimpleGUI` entschieden. Wir hatten auch mit `TKinter` experimentiert, allerdings gefiel uns das Aussehen ohne besondere Einstellungen von `PySimpleGUI` etwas besser.

So weit so knapp, lasst uns gleich zu den Vorarbeiten kommen.

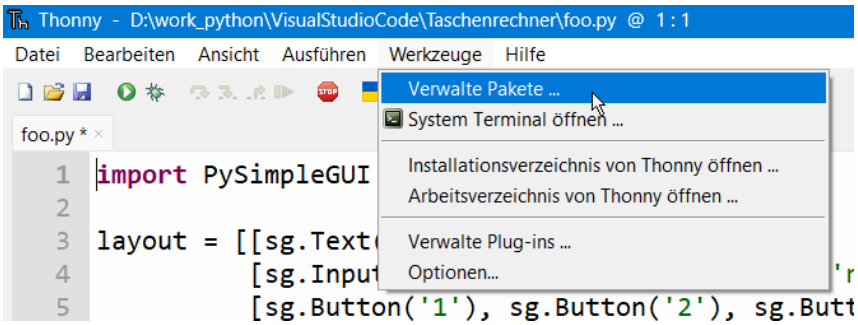
3. Vorarbeiten

Zum vorherigen Projekt gibt es die Vorarbeiten betreffend keine große Veränderung.

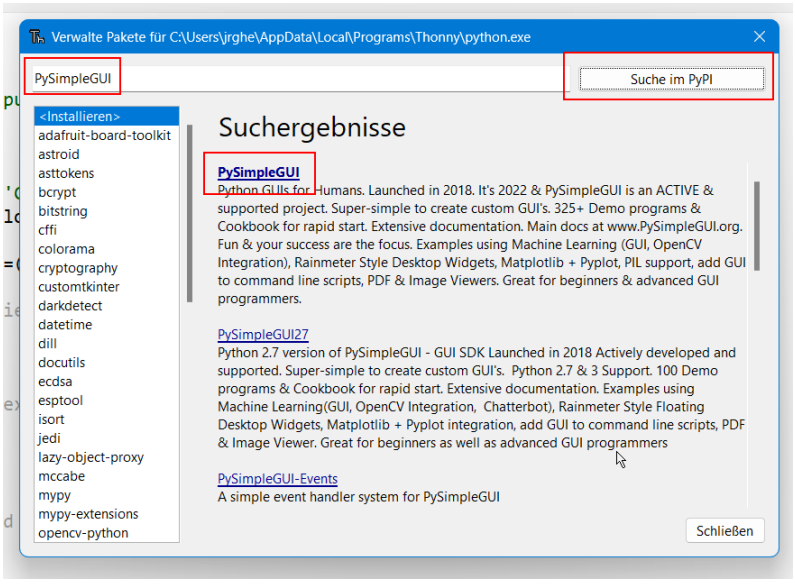
Die Entwicklung habe ich damals in der Thonny IDE gemacht, die werde ich auch hier wieder nutzen. Den Download gibt es auf https://thonny.org/ - zur Entstehung dieses Projektes ist das ist die Version 4.0.2, die habe ich mir jetzt installiert. Diese Version läuft mit Python 3.10.9.

Um die GUI-Entwicklung mit `PySimpleGUI` durchführen zu können, müssen wir das entsprechende Paket dazu noch installieren.

Über den Menüpunkt „Extras/Verwaltete Pakete…“:

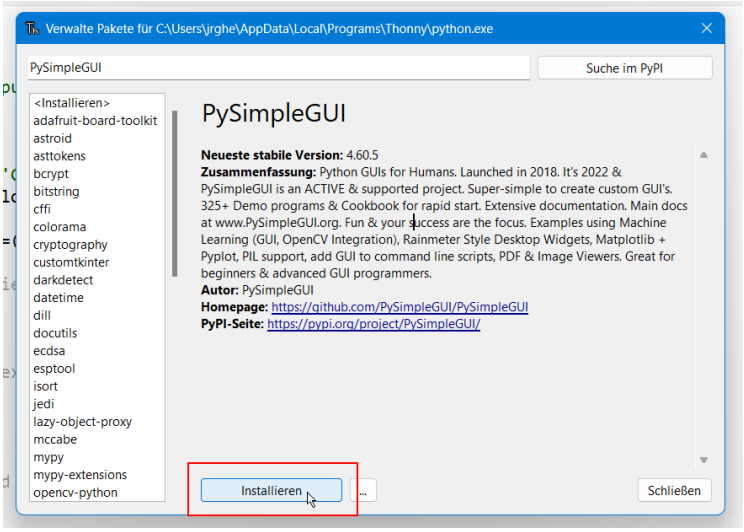


gelangen wir in den Paket-Assistenten:



In der Suchleiste „`PySimpleGUI`“ eingeben, dann „Suchen auf PyPI“ klicken und das Paket `PySimpleGUI` anklicken.

Im folgenden Fenster „Installieren“ klicken.



Nach der Installation einfach auf „Schließen“ klicken.

Das war es dann auch schon mit den Vorbereitungen, genug der Vorarbeiten, los geht’s!

4. Projekt „Taschenrechner"

Wie auch in den anderen Projekten schon praktiziert, machen wir uns zuerst Gedanken über den Funktionsumfang, also die Frage nach dem ‚**Was**'. Das strukturiert das Projekt und wir haben eine klarere Vorstellung, als wenn wir direkt „draufloshacken" würden.

Dann gehen wir das ‚**Wie**' an. Fest steht ja schon, dass wir das mit Python und `PySimpleGUI` lösen wollen.

Im dritten Teil dann kommt die Programmierung dran.

4.1. Der Funktionsumfang

Der Taschenrechner soll neben den Grundrechenarten auch die Eingabe von Klammern ermöglichen, damit heben wir uns von anderen Rechnern ab.

Es soll auch die Möglichkeit bestehen, jeweils die letzte Eingabe rückgängig zu machen.

Mehr ist zu diesem Zeitpunkt zum Funktionsumfang nicht zu sagen. In einer weiteren Ausbaustufe können dann natürlich weitere Funktionen wie Prozentrechnung oder Wurzelziehen hinzugenommen werden, Ihr habt dann das Grundgerüst, dieses selbst zu erweitern.

Damit gleich weiter zum ‚**Wie**'.

4.2. Die Planung der Umsetzung

Dieses Kapitel ist diesmal ziemlich kurz, die meisten Parameter des ‚**Wie**' stehen ja schon fest. Wir nutzen Python als Programmiersprache, als IDE setzen wir wieder Thonny ein, die Oberfläche realisieren wir mit `PySimpleGUI`.

An dieser Stelle würden wir uns Gedanken machen, ob wir zum Beispiel eine Datenbank bräuchten, oder spezielle Anforderungen wie besonders hohe Datenlast oder Hochverfügbarkeit zu berücksichtigen hätten.

Da das bei uns nicht der Fall ist, haben wir hier nichts weiter aufzunehmen.

4.3. Das Coden

Wir werden uns dem Thema in 3 Schritten nähern.

Im ersten Teil nehmen wir uns die Zeit für Datentypen, Teil 2 befasst sich mit Schleifen und Eingaben, Teil 3 ist dann der GUI-Programmierung vorbehalten.

Wer sich mit Datentypen auskennt, sollte Kapitel 4.3.1. überspringen oder nur kurz reinlunzen. Wer auch mit der Funktion input() vertraut ist, kann auch 4.3.2. übergehen und wer sich mit eval() auskennt, sollte direkt zur Kapitel 4.3.4. gehen.

Wie immer gilt, neue oder geänderte Code-Passagen sind zum leichteren Auffinden `grün` hinterlegt, der grau hinterlegte Teil ist dann schon bekannt.

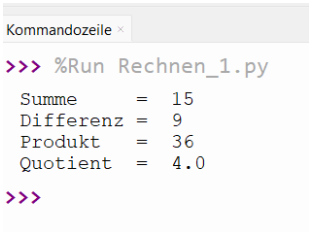
Das waren meine einleitenden Worte zum Thema **Coden**, jetzt legen wir wirklich los!

4.3.1. Datentypen in Python

Wie versprochen, gehen wir es langsam an. Kopiert die nachfolgenden Zeilen in den Editor Eurer Wahl (rechts oben im Quelltext solltet Ihr einen Button finden) und lasst den Code ausführen:

```
1   zahl_1 = 12
2   zahl_2 = 3
3
4   summe = zahl_1 + zahl_2
5   differenz = zahl_1 - zahl_2
6   produkt = zahl_1 * zahl_2
7   quotient = zahl_1 / zahl_2
8
9   print('Summe     = ', summe)
10  print('Differenz = ', differenz)
11  print('Produkt   = ', produkt)
12  print('Quotient  = ', quotient)
13
```

Bei mir kommt dann folgendes heraus:



Was haben wir hier vor uns? In den Zeilen 1 und 2 im Code sind zwei Variablen deklariert, diese sind beide vom Typ „Integer“ (`int`). „Integer“-Werte sind Ganzzahlen, also Zahlen ohne Komma.

Woher weiß ich, dass die Datentypen vom Typ `int` sind? Mit der Funktion `type()` kann man sich den Datentyp einer Variable ausgeben lassen:

```
type(name_der_variablen)
```

Erweitern wir den Code entsprechend, um das auszuprobieren:

```
 1  zahl_1 = 12
 2  zahl_2 = 3
 3
 4  print('Datentyp zahl_1 =', type(zahl_1))
 5  print('Datentyp zahl_2 =', type(zahl_2))
 6
 7  summe = zahl_1 + zahl_2
 8  differenz = zahl_1 - zahl_2
 9  produkt = zahl_1 * zahl_2
10  quotient = zahl_1 / zahl_2
11
12  print('Summe     = ', summe, " ist vom Typ : ", type(summe))
13  print('Differenz = ', differenz, " ist vom Typ : ", type(differenz))
14  print('Produkt    = ', produkt, " ist vom Typ : ", type(produkt))
15  print('Quotient  = ', quotient, " ist vom Typ : ", type(quotient))
```

Die Ausgabe dazu lautet:

```
Kommandozeile
>>> %Run Rechnen_1.py
 Datentyp zahl_1 = <class 'int'>
 Datentyp zahl_2 = <class 'int'>
 Summe     =  15  ist vom Typ :  <class 'int'>
 Differenz =  9  ist vom Typ :  <class 'int'>
 Produkt    =  36  ist vom Typ :  <class 'int'>
 Quotient  =  4.0  ist vom Typ :  <class 'float'>
>>>
```

Was gibt es also noch für Datentypen?

Wenn wir uns die Ausgabe noch einmal betrachten, sehen wir einen Punkt im Wert des Quotienten (`4.0`). Die Variable „`quotient`“ ist also vom Typ „`float`“ oder Fließkommazahl. Wobei das Komma hier kein Komma, sondern ein Punkt ist (das ist in den meisten anglo-amerikanischen Regionen der Fall und da die Syntax von Python auf Englisch beruht, ist das nur konsequent).

Probieren wir es aus. Machen wir uns den Spaß und kopieren nachfolgenden Code in den Editor:

```
 1 string_1 = '1'
 2 string_2 = "zwei"
 3 zahl_mit_komma = 5,2
 4 zahl_mit_punkt = 5.2
 5 zahl_mit_komma_2 = (5,2)
 6 liste_1 = ["Apfel", "Birne", "Zitrone"]
 7 set_1 = {"Apfel", "Badewanne"}
 8 dict_1 = {"deutsch":"Apfel", "english":"apple"}
 9 boolean_1 = True
10
11 print('Datentyp string_1 =', type(string_1))
12 print('Datentyp string_2 =', type(string_2))
13 print('Datentyp zahl_mit_komma =', type(zahl_mit_komma))
14 print('Datentyp zahl_mit_komma_2 =', type(zahl_mit_komma_2))
15 print('Datentyp zahl_mit_punkt =', type(zahl_mit_punkt))
16 print('Datentyp liste_1 =', type(liste_1))
17 print('Datentyp set_1 =', type(set_1))
18 print('Datentyp dict_1 =', type(dict_1))
19 print('Datentyp boolean_1 =', type(boolean_1))
20
```

Was ist das Ergebnis?

```
Kommandozeile
>>> %Run Rechnen_1.py
 Datentyp string_1 = <class 'str'>
 Datentyp string_2 = <class 'str'>
 Datentyp zahl_mit_komma = <class 'tuple'>
 Datentyp zahl_mit_komma_2 = <class 'tuple'>
 Datentyp zahl_mit_punkt = <class 'float'>
 Datentyp liste_1 = <class 'list'>
 Datentyp set_1 = <class 'set'>
 Datentyp dict_1 = <class 'dict'>
 Datentyp boolean_1 = <class 'bool'>
>>>
```

Wir sehen die Typen

`'str'`, `'tuple'`, `'float'`, `'list'`, `'set'`, `'dict'` und `'bool'`.

„`str`“ steht für „String“, also Zeichenketten, die Buchstaben und Zahlen beinhalten können. Diese werden mit den einfachen oder doppelten Anführungsstrichen ausgedrückt.

Die Zahl „5,2“ wird nicht als Zahl, sondern als „`tuple`“ gespeichert. Ein Tupel ist eine Kombination aus mehreren Einträgen, in unserem Fall 2, nämlich die „5“ und die „2“. Wenn wir

`zahl_mit_komma = 5,'zwei'` oder

`zahl_mit_komma = 5,'zwei', 2`

schreiben würden, wäre es also immer noch ein Tupel.

Dann haben wir noch die Typen „`list`“ für Listen, „`set`“ für Sammlungen, „`dict`“ für Dictionary und „`bool`“ für Boolean. Im Verlauf brauchen wir die Listen und die Booleans, daher gehe ich auf die anderen Typen hier nicht weiter ein.

Listen sind ganz praktisch, hier können wir logisch zusammengehörende Informationen ablegen, diese um neue Elemente erweitern oder bestehende Elemente löschen.

Dazu ein kurzer Ausflug:

```
1   zu_kaufen = ["Brot","Milch","Eier","Käse","Wrust"]
2   print("Liste 1 = ", zu_kaufen)
3   zu_kaufen.append("Bananen")
4   print("Liste 2 = ", zu_kaufen)
5   zu_kaufen.sort()
6   print("Liste 3 = ", zu_kaufen)
7   zu_kaufen.remove("Wrust")
8   print("Liste 4 = ", zu_kaufen)
9   zu_kaufen.insert(4,"Wurst")
10  print("Liste 5 = ", zu_kaufen)
11  zu_kaufen.pop()
12  print("Liste 6 = ", zu_kaufen)
13  zu_kaufen.pop(3)
14  print("Liste 7 = ", zu_kaufen)
15  print("Eintrag an 2. Stelle = ", zu_kaufen[1])
16  zu_kaufen.sort(reverse = True)
17  print("Liste 8 = ", zu_kaufen)
```

Die Ausgabe dazu ist

```
Kommandozeile
>>> %Run Rechnen_1.py
 Liste 1 =  ['Brot', 'Milch', 'Eier', 'Käse', 'Wrust']
 Liste 2 =  ['Brot', 'Milch', 'Eier', 'Käse', 'Wrust', 'Bananen']
 Liste 3 =  ['Bananen', 'Brot', 'Eier', 'Käse', 'Milch', 'Wrust']
 Liste 4 =  ['Bananen', 'Brot', 'Eier', 'Käse', 'Milch']
 Liste 5 =  ['Bananen', 'Brot', 'Eier', 'Käse', 'Wurst', 'Milch']
 Liste 6 =  ['Bananen', 'Brot', 'Eier', 'Käse', 'Wurst']
 Liste 7 =  ['Bananen', 'Brot', 'Eier', 'Wurst']
 Eintrag an 2. Stelle =  Brot
 Liste 8 =  ['Wurst', 'Eier', 'Brot', 'Bananen']

>>>
```

Zeile 3: `zu_kaufen.append("Bananen")` # --> fügt ein Element am Ende an
Zeile 5: `zu_kaufen.sort()` # --> sortiert die Liste, geht nur bei gleichen Datentypen!
Zeile 7: `zu_kaufen.remove("Wrust")` # --> löscht das erste Element, das dem Eintrag entspricht
Zeile 9: `zu_kaufen.insert(4,"Wurst")` # --> fügt das Element an der 5. Stelle hinzu (Beginn bei 0!)
Zeile 11: `zu_kaufen.pop()` # --> löscht das letzte Element
Zeile 13: `zu_kaufen.pop(3)` # --> löscht das 4. Element der Liste
Zeile 15: `zu_kaufen[1]` # --> liefert das 2. Element der Liste
Zeile 17: `zu_kaufen.sort(reverse = True)` # --> sortiert absteigend

So viel zu den Listen. Es gäbe zwar noch wesentlich mehr zu den Listen zu sagen, das sprengt aber hier den Rahmen.

Den Typ „`bool`“ kennen wir ja auch schon aus der Steuerung unseres Spiels:

```python
spielen = True
durchgang = 1

print("Start")
while spielen:
    print("Durchgang = ", durchgang)
    if durchgang < 5:
        durchgang += 1
    else:
        spielen = False

print("Ende")
```

Kommandozeile

```
>>> %Run -c $EDITOR_CONTENT
 Start
 Durchgang =  1
 Durchgang =  2
 Durchgang =  3
 Durchgang =  4
 Durchgang =  5
 Ende

>>>
```

Die Variable „`spielen`“ dient uns also zur Unterscheidung, ob wir noch weiterspielen wollen („`spielen`“ hat den Wert „`True`“) oder ob eben nicht („spielen“ hat den Wert „`False`“) und fungiert damit als Schalter.

Zurück zum Rechnen, wie oben beschrieben klappt die Rechnung nur mit Variablen vom Typ `Integer` oder `Float`. `Strings` können nicht berechnet werden. Der nachfolgende Code erzeugt einen Fehler, die beiden Variablen sind als Strings deklariert:

Rechnen_1.py

```python
zahl_1 = '12'
zahl_2 = "3"

print('Datentyp zahl_1 =', type(zahl_1))
print('Datentyp zahl_2 =', type(zahl_2))

summe = zahl_1 + zahl_2
differenz = zahl_1 - zahl_2
produkt = zahl_1 * zahl_2
quotient = zahl_1 / zahl_2

print('Summe     = ', summe, " ist vom Typ : ", type(summe))
print('Differenz = ', differenz, " ist vom Typ : ", type(differenz))
print('Produkt    = ', produkt, " ist vom Typ : ", type(produkt))
print('Quotient  = ', quotient, " ist vom Typ : ", type(quotient))
```

Kommandozeile

```
>>> %Run Rechnen_1.py
 Datentyp zahl_1 = <class 'str'>
 Datentyp zahl_2 = <class 'str'>
Traceback (most recent call last):
  File "D:\work_python\Thonny\Rechner\Rechnen_1.py", line 8, in <module>
    differenz = zahl_1 - zahl_2
TypeError: unsupported operand type(s) for -: 'str' and 'str'

>>>
```

Einzig die Addition funktioniert für Strings, es führt aber zu einer nicht wirklich erwünschten Ausgabe. Kommentieren wir die anderen Rechnungen mal aus:

```
1  zahl_1 = '12'
2  zahl_2 = '3'
3
4  print('Datentyp zahl_1 =', type(zahl_1))
5  print('Datentyp zahl_2 =', type(zahl_2))
6
7  summe = zahl_1 + zahl_2
8  #differenz = zahl_1 - zahl_2
9  #produkt = zahl_1 * zahl_2
10 #quotient = zahl_1 / zahl_2
11
12 print('Summe     = ', summe, " ist vom Typ : ", type(summe))
13 #print('Differenz = ', differenz, " ist vom Typ : ", type(differenz))
14 #print('Produkt    = ', produkt, " ist vom Typ : ", type(produkt))
15 #print('Quotient  = ', quotient, " ist vom Typ : ", type(quotient))
16
17
```

Kommandozeile

```
>>> %Run Rechnen_1.py
 Datentyp zahl_1 = <class 'str'>
 Datentyp zahl_2 = <class 'str'>
 Summe     =  123  ist vom Typ :  <class 'str'>

>>>
```

Man sieht, das Ergebnis ist „`123`“, wobei es nicht die Zahl 123 ist, sondern die Zeichenkette „`12`“ und das Zeichen „`3`“ hintereinander ist.

Klarer wird es mit folgender Kombination:

```
1  zahl_1 = '12'
2  zahl_2 = 'drei'
3
4  print('Datentyp zahl_1 =', type(zahl_1))
5  print('Datentyp zahl_2 =', type(zahl_2))
6
7  summe = zahl_1 + zahl_2
8  #differenz = zahl_1 - zahl_2
9  #produkt = zahl_1 * zahl_2
10 #quotient = zahl_1 / zahl_2
11
12 print('Summe     = ', summe, " ist vom Typ : ", type(summe))
13 #print('Differenz = ', differenz, " ist vom Typ : ", type(differenz))
14 #print('Produkt    = ', produkt, " ist vom Typ : ", type(produkt))
15 #print('Quotient  = ', quotient, " ist vom Typ : ", type(quotient))
16
17
```

Kommandozeile

```
>>> %Run Rechnen_1.py
 Datentyp zahl_1 = <class 'str'>
 Datentyp zahl_2 = <class 'str'>
 Summe     =  12drei  ist vom Typ :  <class 'str'>

>>>
```

Das Ergebnis ist jetzt der String „`12drei`“. Damit kann natürlich nicht gerechnet werden. Um die Berechnung mit dem Typ `Float` zu demonstrieren, können wir aus der „`12`“ eine „`12.0`“ machen, das Ergebnis ist dann:

```python
zahl_1 = 12.0
zahl_2 = 3

print('Datentyp zahl_1 =', type(zahl_1))
print('Datentyp zahl_2 =', type(zahl_2))

summe = zahl_1 + zahl_2
differenz = zahl_1 - zahl_2
produkt = zahl_1 * zahl_2
quotient = zahl_1 / zahl_2

print('Summe     = ', summe, " ist vom Typ : ", type(summe))
print('Differenz = ', differenz, " ist vom Typ : ", type(differenz))
print('Produkt    = ', produkt, " ist vom Typ : ", type(produkt))
print('Quotient  = ', quotient, " ist vom Typ : ", type(quotient))
```

Kommandozeile

```
>>> %Run Rechnen_1.py
 Datentyp zahl_1 = <class 'float'>
 Datentyp zahl_2 = <class 'int'>
 Summe     =  15.0  ist vom Typ :  <class 'float'>
 Differenz =  9.0  ist vom Typ :  <class 'float'>
 Produkt    =  36.0  ist vom Typ :  <class 'float'>
 Quotient  =  4.0  ist vom Typ :  <class 'float'>

>>>
```

Damit sind jetzt alle Ergebnisvariablen auch vom Typ `Float`.

Das soll es dann auch zum Thema Datentypen gewesen sein.

4.3.2. Eingabe von Zahlen

Für einen echten Taschenrechner brauchen wir die Möglichkeit einer Eingabe von 2 oder mehr Zahlen. Eingaben in Python werden mit der Funktion `input()` realisiert. Nehmen wir wieder den Code mit der String-Ausgabe und ergänzen ihn in den Zeilen 1 und 2:

```
 1 zahl_1 = input('1. Zahl eingeben: ')
 2 zahl_2 = input('2. Zahl eingeben: ')
 3
 4 print('Datentyp zahl_1 =', type(zahl_1))
 5 print('Datentyp zahl_2 =', type(zahl_2))
 6
 7 summe = zahl_1 + zahl_2
 8 #differenz = zahl_1 - zahl_2
 9 #produkt = zahl_1 * zahl_2
10 #quotient = zahl_1 / zahl_2
11
12 print('Summe      = ', summe, " ist vom Typ : ", type(summe))
13 #print('Differenz = ', differenz, " ist vom Typ : ", type(differenz))
14 #print('Produkt    = ', produkt, " ist vom Typ : ", type(produkt))
15 #print('Quotient  = ', quotient, " ist vom Typ : ", type(quotient))
```

Die Funktion führt dazu, dass zuerst auf die Eingabe der `zahl_1` gewartet wird (muss man unten in der Kommandozeile eingeben und mit Enter bestätigen), wenn diese erfolgt ist, wird auf die Eingabe von `zahl_2` gewartet:

```
Kommandozeile
     %Run Rechnen_1.py
 1. Zahl eingeben:
```

```
Kommandozeile
>>> %Run Rechnen_1.py
 1. Zahl eingeben:  42
 2. Zahl eingeben:
```

Wie man in der Ausgabe sieht, sind beide Zahlen nun wieder vom Typ String, genau wie die Summe:

```
Kommandozeile
>>> %Run Rechnen_1.py
 1. Zahl eingeben: 42
 2. Zahl eingeben: 7
 Datentyp zahl_1 = <class 'str'>
 Datentyp zahl_2 = <class 'str'>
 Summe      =  427  ist vom Typ :  <class 'str'>
>>>
```

Eingaben sind aber immer vom Typ String! Wie können wir aber damit rechnen?

Dafür gibt es weitere Funktionen, deren Anwendung sich „casting“ nennt. Dabei weist man das Programm an, zum Beispiel einen String in einen Integer-Wert umzuwandeln. Aber auch andere „castings“ sind möglich. In unserem Fall wandeln wir aber nur einen String in einen Integer um.

Das geschieht, in dem die `input()`-Funktion in die `int()`-Funktion eingebunden wird:

```
1 zahl_1 = int(input('1. Zahl eingeben: '))
2 zahl_2 = int(input('2. Zahl eingeben: '))
3 print('Datentyp zahl_1 =', type(zahl_1))
4 print('Datentyp zahl_2 =', type(zahl_2))
5 …
```

Das Ergebnis ist jetzt, dass `zahl_1` und `zahl_2` vom Typ Integer sind:

```
Kommandozeile
>>> %Run Rechnen_1.py
 1. Zahl eingeben: 42
 2. Zahl eingeben: 7
 Datentyp zahl_1 = <class 'int'>
 Datentyp zahl_2 = <class 'int'>
 Summe     =  49  ist vom Typ :  <class 'int'>
>>>
```

Damit funktioniert die Berechnung wieder für alle Rechenarten:

```
1  zahl_1 = int(input('1. Zahl eingeben: '))
2  zahl_2 = int(input('2. Zahl eingeben: '))
3
4  print('Datentyp zahl_1 =', type(zahl_1))
5  print('Datentyp zahl_2 =', type(zahl_2))
6
7  summe = zahl_1 + zahl_2
8  differenz = zahl_1 - zahl_2
9  produkt = zahl_1 * zahl_2
10 quotient = zahl_1 / zahl_2
11
12 print('Summe     = ', summe, " ist vom Typ : ", type(summe))
13 print('Differenz = ', differenz, " ist vom Typ : ", type(differenz))
14 print('Produkt    = ', produkt, " ist vom Typ : ", type(produkt))
15 print('Quotient  = ', quotient, " ist vom Typ : ", type(quotient))
16
```

```
Kommandozeile
>>> %Run Rechnen_1.py
 1. Zahl eingeben: 42
 2. Zahl eingeben: 7
 Datentyp zahl_1 = <class 'int'>
 Datentyp zahl_2 = <class 'int'>
 Summe     =  49  ist vom Typ :  <class 'int'>
 Differenz =  35  ist vom Typ :  <class 'int'>
 Produkt    =  294  ist vom Typ :  <class 'int'>
 Quotient  =  6.0  ist vom Typ :  <class 'float'>
>>>
```

Noch ein Hinweis, geben wir statt einer Zahl irgendwelche Buchstaben ein, bricht das Programm mit einer Fehlermeldung ab:

```
Kommandozeile
>>> %Run Rechnen_1.py
 1. Zahl eingeben: gehen
 Traceback (most recent call last):
   File "D:\work_python\Thonny\Rechner\Rechnen_1.py", line 1, in <module>
     zahl_1 = int(input('1. Zahl eingeben: '))
 ValueError: invalid literal for int() with base 10: 'gehen'
>>>
```

Gleiches gilt aber leider auch für die Eingabe von Fließkommazahlen:

```
Kommandozeile
>>> %Run Rechnen_1.py
 1. Zahl eingeben: 12.0
 Traceback (most recent call last):
   File "D:\work_python\Thonny\Rechner\Rechnen_1.py", line 1, in <module>
     zahl_1 = int(input('1. Zahl eingeben: '))
 ValueError: invalid literal for int() with base 10: '12.0'
>>>
```

Okay, das können wir erst einmal verschmerzen. Sonst sieht das ja schon ganz gut aus, das Programm wird aber immer nur einmal ausgeführt, nach Abschluss der Berechnung schließt es sich.

Erweitern wir unser Programm also um eine Schleife und lassen wir auch gleich das Rechenzeichen mit einlesen:

```
1  # Programmstart
2  berechnen = True
3
4  while berechnen:
5      zahl_1 = int(input('1. Zahl eingeben: '))
6      zahl_2 = int(input('2. Zahl eingeben: '))
7      rechenzeichen = input('Rechenzeichen   : ')
8
9      if rechenzeichen == '+':
10         summe = zahl_1 + zahl_2
11         print('Summe          = ', summe)
12     elif rechenzeichen == '-':
13         differenz =  zahl_1 - zahl_2
14         print('Differenz      = ', differenz)
15     elif rechenzeichen == '*':
16         prdukt = zahl_1 * zahl_2
17         print('Produkt        = ', produkt)
18     elif rechenzeichen == '/':
19         quotient = zahl_1 / zahl_2
20         print('Quotient       = ', quotient)
21     elif rechenzeichen == 'x':
22         break
23     else:
24         print('Rechenzeichen nicht erkannt')
25         berechnen = False
26
```

Zeile 2 führt eine boolesche Variable ein, wir erinnern uns, eine Variable, die nur 2 Zustände hat, „`True`“ also „an“ oder „`False`“ oder „aus“.

Die `while`-Schleife wird solange wieder und wieder durchlaufen, bis sie auf „`False`“ gesetzt wird. Das geschieht dann, wenn das Rechenzeichen nicht „+“, „-“, „*“, „/“ oder „x“ ist.

Bei Eingabe der Grundrechenarten wird die jeweilige Berechnung durchgeführt, bei Eingabe von „x“ wird die Schleife mit dem Befehl „`break`“ verlassen.

Geben wir zum Beispiel ein „n“ ein, erfolgt die Ausgabe der Fehlermeldung und das Programm wird beendet, indem der Schalter „`berechnen`“ auf „`False`“ gesetzt wird.

Soweit verständlich, oder?

4.3.3. Rechnen mit der Funktion `eval()`

Bevor wir uns nun der GUI-Programmierung zuwenden, habe ich noch eine letzte Änderung, die allerdings nichts mit dem zu tun hat, was wir bisher gemacht haben.

So sieht die Änderung aus:

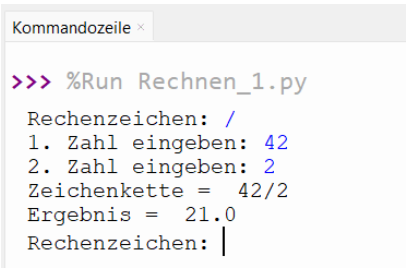
```
 1 berechnen = True
 2
 3 while berechnen:
 4     rechenzeichen = input('Rechenzeichen: ')
 5
 6     if rechenzeichen == 'x':
 7         break
 8
 9     zahl_1 = int(input('1. Zahl eingeben: '))
10     zahl_2 = int(input('2. Zahl eingeben: '))
11
12     zeichenkette = f"{zahl_1}{rechenzeichen}{zahl_2}"
13     print('Zeichenkette = ', zeichenkette)
14
15     ergebnis = eval(zeichenkette)
16     print('Ergebnis = ', ergebnis)
17
```

Das muss ich erklären. Die entscheidende Stelle ist in Zeile 15 zu finden. Es gibt in Python die Funktion `eval()`, die eine Zeichenkette „untersucht“ und dabei das Ergebnis ermittelt, falls diese Zeichen enthält, die sich auch berechnen lassen.

Die Zeichenkette wird mit einem `f-string` (das steht für „formatted string literal”) in Zeile 12 zusammengesetzt. Das ist ein sehr nützliches Werkzeug, mit dem man sich beschäftigen sollte!

Die Eingabe des Rechenzeichens ist nach Zeile 4 gewandert, damit ist es das erste Zeichnen, das abgefragt wird. Wenn also ein „x“ eingegeben wird, wird die Schleife mit dem Befehl „`break`“ verlassen (Zeile 7).

Die Ausgabe sieht dann bei mir so aus:

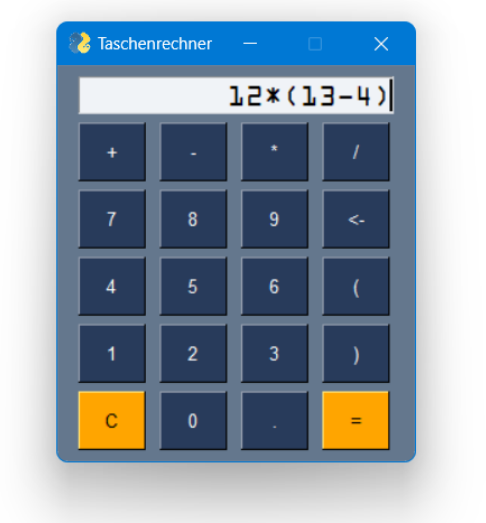


Hand aufs Herz, wer von Euch hat sich gefragt „Echt jetzt? Das war’s? Wozu haben wir die ganzen Vor-Versionen mit Datentypen rauf und runter gebaut?“. Die Antwort ist einfach - wichtig ist doch, dass wir uns mit Python beschäftigt haben 😉

Damit haben wir das Vorgeplänkel hinter uns, wenden wir uns nun der GUI-Programmierung zu.

4.3.4. Das Fenster mit `PySimpleGui`

Werfen wir zunächst noch einmal einen Blick auf das Bild in Kapitel 2:



So soll unser Taschenrechner also aussehen.

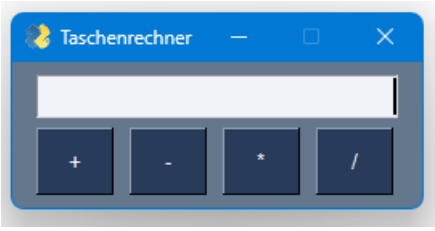
Wir können 20 Schaltflächen (oder englisch „button“) erkennen, 4 in jeder Reihe und 5 Reihen.

Darüber, sozusagen in der ersten Reihe, ist das Eingabefeld platziert. Hier werden die Zahlen und Rechenzeichen angezeigt.

In `PySimpleGui` werden die einzelnen Elemente im „`layout`“ gebündelt. Damit wir uns der Materie sachte nähern, schauen wir uns die Umsetzung mal nur für die ersten beiden Zeilen an. Der Code lautet wie folgt:

```
1  import PySimpleGUI as sg
2
3  # die Elemente im Fenster werden im layout gebündelt:
4  layout = [[sg.Input(size=(17, 1), justification='right', key='eingabe',
5                      font=('OCR A Extended', 16))],
6            [sg.Button('+'), sg.Button('-'), sg.Button('*'), sg.Button('/')]
7          ]
8
9  # das Fenster selber wird erzeugt
10 window = sg.Window('Taschenrechner', layout, default_button_element_size=(5,2),
11                     auto_size_buttons=False)
12
13 # Programmschleife
14 while True:
15     event, values = window.read()
16     if event == sg.WIN_CLOSED:
17         break
18
```

Wenn Ihr das als Programm ausführen lasst, sollte der Rechner etwa so aussehen:



Gehen wir den Code durch. Zeile 1 importiert die Bibliothek. Voraussetzung für die Nutzung ist natürlich, dass Ihr sie installiert habt.

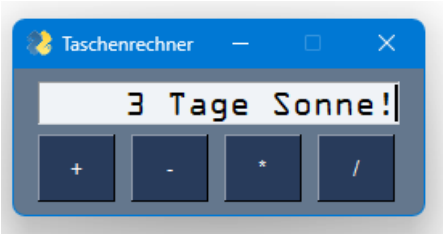
Zeile 4 bis Zeile 7 ist das oben angekündigte „`layout`“. Das schauen wir uns gleich näher an.

In Zeile 10 wird das Fenster selber instanziiert. Der erste Parameter in der Funktion ist der Name, der oben links angezeigt wird (`'Taschenrechner'`). Parameter 2 ist der Verweis auf das `layout`, in unserem Fall haben wir es auch genau so benannt, nämlich `layout`. Der an dritter Stelle stehende Parameter „`default_button_element_size=(5,2)`“ gibt an, dass alle Button (ich nehme jetzt mal die denglische Form der Schaltfläche) gleich groß sein sollen. Der Wert „`auto_size_buttons=False`“ besagt, dass sich die Button nicht nach der Anzahl der Stallen richten soll, sondern, dass die Größe fix bleibt. Setzt den Parameter auch gerne mal auf „`True`“, dann seht Ihr den Unterschied.

Ab Zeile 14 ist die Programmschleife codiert. In Zeile 15 wird das Ereignis innerhalb des Fensters (das „`event`“, also z.B. das Klicken auf einen Button) abgefragt, aber auch der Wert oder Zustand des Objekts („`value`“), also bei gedrücktem ‚+‘-Button das ‚+‘ selbst.

Besonders wird der Klick auf das „X“ in der rechten oberen Ecke des Fensters geachtet, das schließt das Programm, indem die Schleife mit „`break`“ verlassen wird.

Wenn Ihr jetzt etwas über die Tatstatur eingebt, seht Ihr die Zeichen in der Anzeige.



Damit ist natürlich noch keine Berechnung möglich, dazu kommen wir aber später.

Schauen wir jetzt genauer auf die Zeilen 4 bis 7, also das „`layout`“.

```
2 …
3 # die Elemente im Fenster werden im layout gebündelt:
4 layout = [[sg.Input(size=(17, 1), justification='right', key='eingabe',
5                     font=('OCR A Extended', 16))],
6           [sg.Button('+'), sg.Button('-'), sg.Button('*'), sg.Button('/')]
7         ]
8 …
```

Wir haben zwei ineinander verschachtelte Listen. Das `layout` selbst sieht so aus:

```
layout = []
```

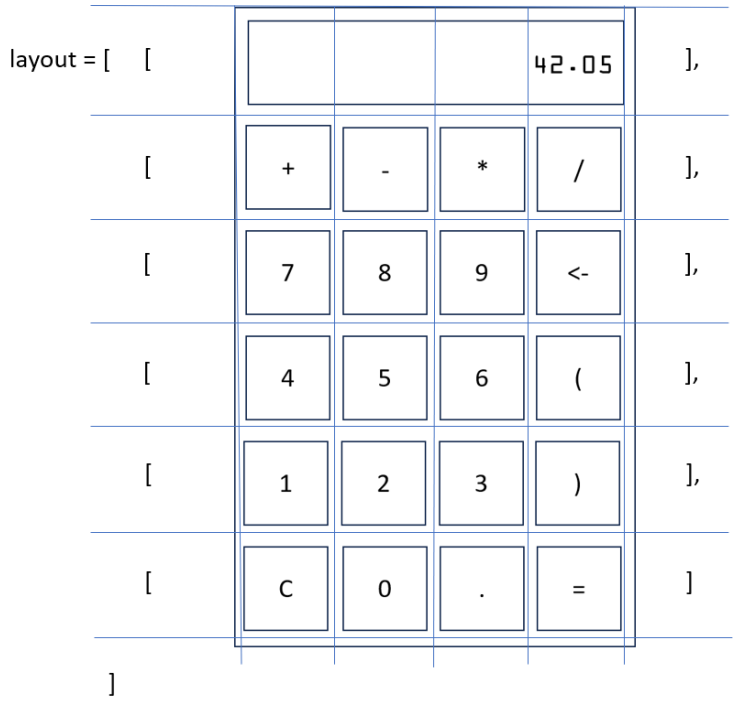
Es erstreckt sich über 3 Zeilen. Innerhalb von `layout` gibt es nun eine Liste für die erste Zeile…:

```
[sg.Input(size=(17, 1), …, font=('OCR A Extended', 16))]
```

…und eine Liste für die zweite Zeile, in der die Button enthalten sind:

```
[sg.Button('+'), sg.Button('-'), sg.Button('*'), sg.Button('/')]
```

Um alle relevanten Button für den Taschenrechner zu implementieren, müsste die Erweiterung dem folgenden Bild entsprechend gemacht werden:



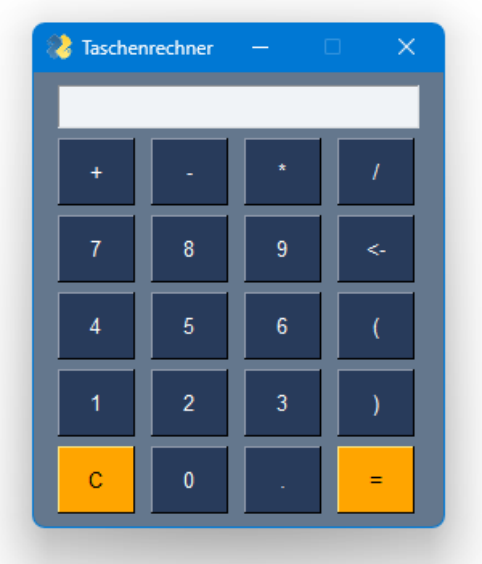
Ich habe da mal was vorbereitet…

```
1  import PySimpleGUI as sg
2
3  # die Elemente im Fenster werden im layout gebündelt:
4  layout = [[sg.Input(size=(17, 1), justification='right', key='eingabe',
5                      font=('OCR A Extended', 16))],
6            [sg.Button('+'), sg.Button('-'), sg.Button('*'), sg.Button('/')],
7            [sg.Button('7'), sg.Button('8'), sg.Button('9'), sg.Button('<-')],
8            [sg.Button('4'), sg.Button('5'), sg.Button('6'), sg.Button('(')],
9            [sg.Button('1'), sg.Button('2'), sg.Button('3'), sg.Button(')')],
10           [sg.Button('C', button_color=("black","orange")), sg.Button('0'),
11            sg.Button('.'), sg.Button('=', button_color=("black","orange"))]
12         ]
13
14 # das Fenster selber wird erzeugt
15 window = sg.Window('Taschenrechner', layout, default_button_element_size=(5,2),
16                    auto_size_buttons=False)
17
18 # Programmschleife
19 while True:
20     event, values = window.read()
21     if event == sg.WIN_CLOSED:
22         break
23
```

Achtet bitte darauf, dass Zeilen 10 und 11 eigentlich zusammengehören, ich habe der besseren Lesbarkeit halber einen Zeilenumbruch eingefügt.

Die Anweisungen ‚`button_color=("black","orange")`‘ in den Zeilen 10 und 11 geben dem Ruhezustand des Button die Farbe „`orange`“ mit, sobald er angeklickt wird, ändert sich die Farge auf Schwarz. Spielt hier auch gerne mal mit unterschiedlichen Farben herum.

Wenn Ihr das jetzt laufen lasst, sollte das dann so aussehen:



Hübsch, oder?

Für die Codierung der Button nehmen wir uns mal wieder ein neues Unterkapitel vor.

4.3.5. Die Button

Schauen wir uns nochmal Zeile 16 aus dem letzten Code-Stand vor der Einführung der PySimpleGUI an (am Ende von Kapitel 4.3.1.):

```
1  # Programmstart
2  berechnen = True
3
4  while berechnen:
5      rechenzeichen = input('Rechenzeichen: ')
6
7      if rechenzeichen == 'x':
8          break
9
10     zahl_1 = int(input('1. Zahl eingeben: '))
11     zahl_2 = int(input('2. Zahl eingeben: '))
12
13     zeichenkette = (f"{zahl_1}{rechenzeichen}{zahl_2}")
14     print('Zeichenkette = ', zeichenkette)
15
16     ergebnis = eval(zeichenkette)
17     print('Ergebnis = ', ergebnis)
18
```

Wir müssen also dafür sorgen, dass wir alle Zeichen hintereinander in *einer* Zeichenkette speichern. Das gibt uns dann die Möglichkeit, die `eval()`-Funktion aufzurufen, wenn wir auf den ‚='-Button klicken.

Also sorgen wir zuerst dafür, dass wir eine Zeichenkette implementieren, die immer dann gefüllt wird, wenn auf irgendeinen Button außer den 3 Funktionsbutton ‚C', ‚=' oder ‚<-' geklickt wird. Der Code dafür ist denkbar einfach:

```
17 …
18 # Programmschleife
19 while True:
20     event, values = window.read()
21     if event == sg.WIN_CLOSED:
22         break
23     if event in '1234567890()+-*/.':
24         zeichenkette = values['eingabe']
25         zeichenkette += event
26         window['eingabe'].update(zeichenkette)
27
```

Wir hängen in der Programmschleife also eine weitere `if`-Bedingung dran. Dieser Teil wird immer dann durchlaufen, wenn das `event` eines der in der `in`-Abfrage enthaltenen Zeichen ist. Wenn der Button ‚8' gedrückt wird, wird auch der Wert ‚8' übergeben.

Der String ‚`zeichenkette`' wird also in Zeile 24 aus dem Eingabefeld gelesen. Wenn sie noch leer ist, steht halt nichts drin. In Zeile 25 wird der Wert des gedrückten Button an den leeren oder bereits gefüllten String ‚`zeichenkette` ' angehängt.

In Zeile 26 wird das Feld ‚`eingabe`' mit dem neuen String aktualisiert. Fertig.

Jetzt kommt die Behandlung der anderen Button.

Kümmern wir uns zuerst um das ‚C'. Das ist nun wirklich einfach, wenn das ‚C' gedrückt wird, müssen wir nur die Zeichenkette löschen oder leeren, dazu erweitern wir wieder die Programmschleife:

```
17 …
18 # Programmschleife
19 while True:
20     event, values = window.read()
21     if event == sg.WIN_CLOSED:
22         break
23     if event in '1234567890()+-*/.':
24         zeichenkette = values['eingabe']
25         zeichenkette += event
26         window['eingabe'].update(zeichenkette)
27     if event == 'C':
28         zeichenkette = ''
29         window['eingabe'].update(zeichenkette)
```

Nichts Überraschendes, oder?

Wenden wir uns nun dem ‚<-'-Button zu. Was wollen wir erreichen? Das letzte eingegebene Zeichen, also der `value` des zuletzt geklickten Button) soll gelöscht werden. Hier müssen wir uns eines Tricks bedienen:

```
17 …
18 # Programmschleife
19 while True:
20     …
30     if event == '<-':
31         laenge = len(zeichenkette)
32         zeichenkette = zeichenkette[:laenge-1]
33         window['eingabe'].update(zeichenkette)
34
```

Wir lassen uns die Länge der Zeichenkette mittels Funktion `len()` errechnen und speichern diese in der neuen Variable `laenge` (Zeile 31).

In Zeile 32 übernehmen wir in `zeichenkette` nur die ersten Stellen ohne die letzte Stelle (`lange-1`). Haben wir also zum Beispiel `(12+3)*21` eingegeben, ist der Wert von ‚`laenge`' ‚9'. Nach dem Klick auf den ‚<-'-Button wird die `zeichenkette` bis zur Stelle ‚8' gefüllt, es bleibt also der Wert `(12+3)*2` bestehen.

In Zeile 33 wird die Eingabe mit dem neuen Wert dann überschrieben.

Damit kommen wir zum ‚='-Button. Die einfachste Umsetzung wäre:

```
17 …
18 # Programmschleife
19 while True:
20     …
34     if event == '=':
35         ergebnis = eval(zeichenkette)
36         window['eingabe'].update(ergebnis)
37
```

ABER:

Wenn Ihr das umgesetzt habt, gebt mal bitte einen „nicht rechenbaren" String wie `(12+3/*21` ein. Was passiert?

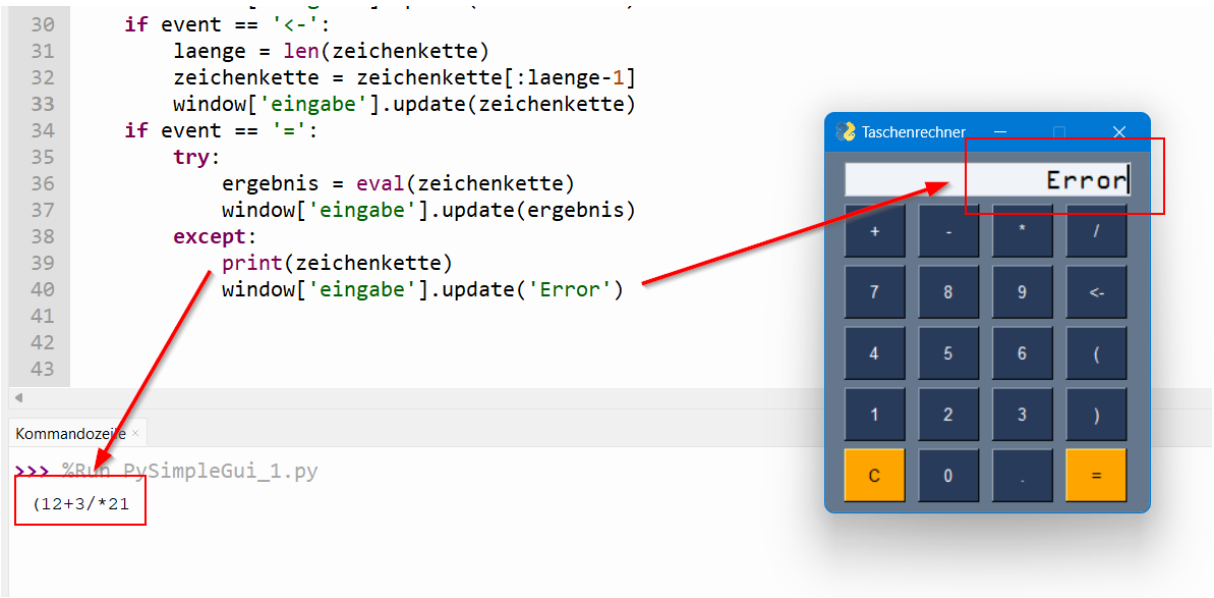
Richtig, das Programm geht kaputt mit der Fehlermeldung:

```
Kommandozeile
>>> %Run PySimpleGui_1.py
  Traceback (most recent call last):
    File "D:\work python\VisualStudioCode\Taschenrechner\finale Stände\PySimpleGui 1.py", line 37, in <module>
      ergebnis = eval(zeichenkette)
    File "<string>", line 1
      (12+3/*21
            ^
  SyntaxError: invalid syntax

>>>
```

Was ist also zu tun? Wir erweitern die `if`-Anweisung um einen `try/except`-Block:

```
17  …
18  # Programmschleife
19  while True:
20      …
34      if event == '=':
35          try:
36              ergebnis = eval(zeichenkette)
37              window['eingabe'].update(ergebnis)
38          except:
39              print(zeichenkette)
40              window['eingabe'].update('Error')
41
```

Wenn wir das jetzt laufen lassen, sehen wir die nicht verarbeitbare Zeichenkette in der Kommandozeile, in der ‚`eingabe`' wird der String ‚`Error`' angezeigt, das Programm stürzt aber nicht ab:



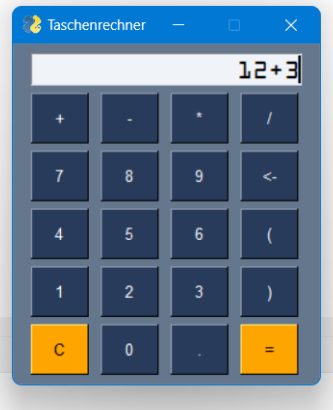
Damit sind wir jetzt schon ganz gut aufgestellt. Bis hierhin ist soweit alles klar?

Sehr schön. Damit kommen wir von meiner Seite zur letzten Aufgabe am Taschenrechner. Wir haben bis jetzt alle Button bearbeitet, was der Taschenrechner aber noch nicht kann, ist eine Eingabe über die Tatstatur korrekt zu verarbeiten.

Was passiert bei der Eingabe von „12+3“ über die Tastatur, wenn wir dann den ‚=‘-Button drücken:

Es kommt zu einer Fehlermeldung:

```
31          laenge = len(zeichenkette)
32          zeichenkette = zeichenkette[:laenge-1]
33          window['eingabe'].update(zeichenkette)
34      if event == '=':
35          try:
36              ergebnis = eval(zeichenkette)
37              window['eingabe'].update(ergebnis)
38          except:
39              print(zeichenkette)
40              window['eingabe'].update('Error')
41
42
43
```



Kommandozeile

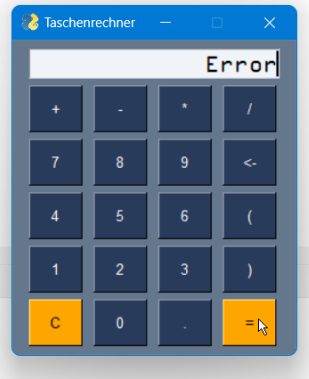
```
>>> %Run PySimpleGui_1.py
  Traceback (most recent call last):
    File "D:\work python\VisualStudioCode\Taschenrechner\finale Stände\PySimpleGui_1.py", line 36, in <module>
      ergebnis = eval(zeichenkette)
  NameError: name 'zeichenkette' is not defined

  During handling of the above exception, another exception occurred:

  Traceback (most recent call last):
    File "D:\work python\VisualStudioCode\Taschenrechner\finale Stände\PySimpleGui_1.py", line 39, in <module>
      print(zeichenkette)
  NameError: name 'zeichenkette' is not defined

>>>
```

Den Fehler können wir beheben, indem wir die Variable `zeichenkette` außerhalb der Schleife definieren:

```
16
17  zeichenkette = ''
18
19  # Programmschleife
20  while True:
21      event, values = window.read()
22      if event == sg.WIN_CLOSED:
```

Besser, aber dann kommt immer noch der String ‚`Error`‘:

```
16
17  zeichenkette = ''
18
19  # Programmschleife
20  while True:
21      event, values = window.read()
22      if event == sg.WIN_CLOSED:
23          break
24      if event in '1234567890()+-*/.':
25          zeichenkette = values['eingabe']
26          zeichenkette += event
27          window['eingabe'].update(zeichenkette)
28      if event == 'C':
29          zeichenkette = ''
30          window['eingabe'].update(zeichenkette)
```



Kommandozeile

```
>>> %Run PySimpleGui_1.py
```

Aber warum ist das so? Wo liegt der Fehler? Wir haben dem Programm nicht mitgeteilt, dass es auch auf die Eingabe der Tastatur reagieren soll. Das müssen wir noch nachholen.

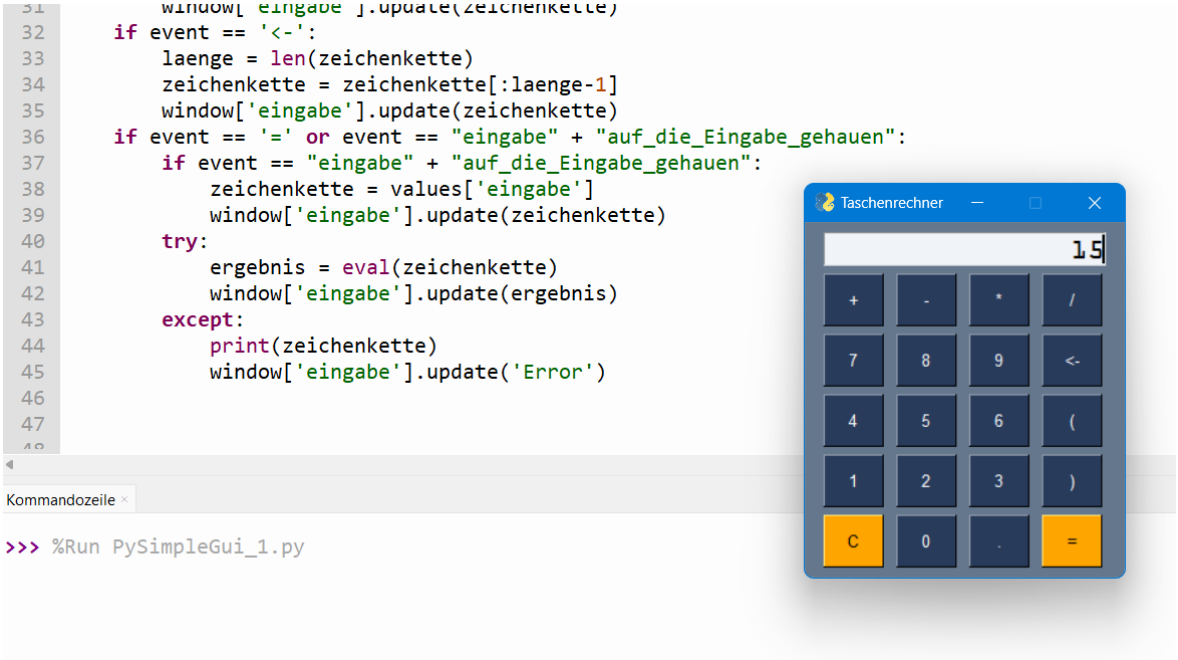
Dazu müssen wir an der Instanziierung des Fensters 2 Änderungen vornehmen. Ich habe jetzt mal den finalen Code vollständig aufgelistet:

```
 1 import PySimpleGUI as sg
 2 # die Elemente im Fenster
 3 layout = [[sg.Input(size=(17, 1), justification='right', key='eingabe',
 4                     font=('OCR A Extended', 16))],
 5           [sg.Button('+'), sg.Button('-'), sg.Button('*'), sg.Button('/')],
 6           [sg.Button('7'), sg.Button('8'), sg.Button('9'), sg.Button('<-')],
 7           [sg.Button('4'), sg.Button('5'), sg.Button('6'), sg.Button('(')],
 8           [sg.Button('1'), sg.Button('2'), sg.Button('3'), sg.Button(')')],
 9           [sg.Button('C', button_color=("black","orange")), sg.Button('0'),
10            sg.Button('.'), sg.Button('=', button_color=("black","orange"))]
11         ]
12
13 # das Fenster selber wird erzeugt
14 window = sg.Window('Taschenrechner', layout, default_button_element_size=(5,2),
15                    auto_size_buttons=False, finalize=True)
16 window['eingabe'].bind("<Return>", "auf_die_Eingabe_gehauen")
17
18 zeichenkette = ''
19
20 # Programmschleife
21 while True:
22     event, values = window.read()
23     if event == sg.WIN_CLOSED:
24         break
25     if event in '1234567890()+-*/.':
26         zeichenkette = values['eingabe']
27         zeichenkette += event
28         window['eingabe'].update(zeichenkette)
29     if event == 'C':
30         zeichenkette = ''
31         window['eingabe'].update(zeichenkette)
32     if event == '<-':
33         laenge = len(zeichenkette)
34         zeichenkette = zeichenkette[:laenge-1]
35         window['eingabe'].update(zeichenkette)
36     if event == '=' or event == "eingabe" + "auf_die_Eingabe_gehauen":
37         if event == "eingabe" + "auf_die_Eingabe_gehauen":
38             zeichenkette = values['eingabe']
39             window['eingabe'].update(zeichenkette)
40         try:
41             ergebnis = eval(zeichenkette)
42             window['eingabe'].update(ergebnis)
43         except:
44             print(zeichenkette)
45             window['eingabe'].update('Error')
46
```

Damit der Taschenrechner für die Eingabe bereit ist, können wir das Fenster aktivieren, das machen wir mit der Erweiterung ‚`, finalize=True` ‘ in Zeile 15. In Zeile 16 „binden“ wir `<Return>` an das Fenster, sodass wir eine Eingabe verarbeiten können, die mit Enter bestätigt wurde.

Zeile 36 erweitern wir um die event-Abfrage auf die Eingabe mit Enter und wenn die Eingabe über die Tatstatur erfolgt ist, lesen wir zuerst den aktuellen Wert von ‚`eingabe`‘ und füllen damit ‚`zeichenkette`‘.

Damit ist die Eingabe von ‚12+3' über die Tastatur mit abschließender Enter-Taste erfolgreich:



Sieht bei Euch auch so aus? Dann von mir ein donnerndes „Bravo!", das habt Ihr gut gemacht, von meiner Seite war es das mit dem Coden.

5. Abschluss

So, damit bin ich auch schon wieder am Ende des Projekts angelangt. Ich hoffe, es hat Euch ebenso viel Spaß gemacht wie mir. Falls ich irgendwo etwas nicht tief genug oder unverständlich erklärt haben sollte, lasst es mich gerne wissen.

Was gäbe es noch zu tun? Raum für Verbesserungen gibt es immer.

Ihr könntet jetzt vielleicht noch Prozentrechnung oder Berechnung von Winkelfunktionen hinzufügen. Dazu solltet Ihr Euch die `math`-Bibliothek mal ansehen.

Ein Tipp an dieser Stelle, 2^3 könnt Ihr schon jetzt berechnen lassen, das sind macht Ihr mit 2 Sternen, also ‚`2**3`'.

Ihr könntet Euch auch die letzten 10 Rechenschritte in einer Textdatei merken und diese wieder zur Anzeige anbieten.

Damit lasse ich Euch mit Eurem neugewonnenen oder aufgefrischten Können wieder alleine.

Ansonsten gilt wie immer, viel Spaß beim Nachbauen und wenn Euch was Tolles einfallen sollte, was man umsetzen könnte, schickt mir gerne eine Mail an papa@papa-programmiert.de, ich würde mich sehr freuen, genau wie auch über eine Rückmeldung zu diesem Dokument.

Viel Spaß weiterhin beim Coden, bleibt neugierig und hartnäckig.

Viele Grüße, Papa