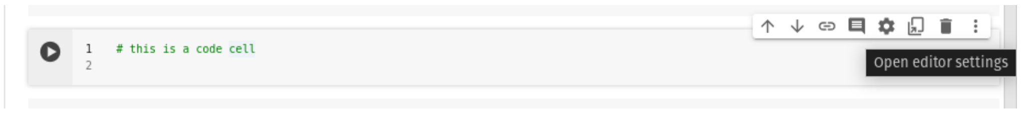
1. editor settings

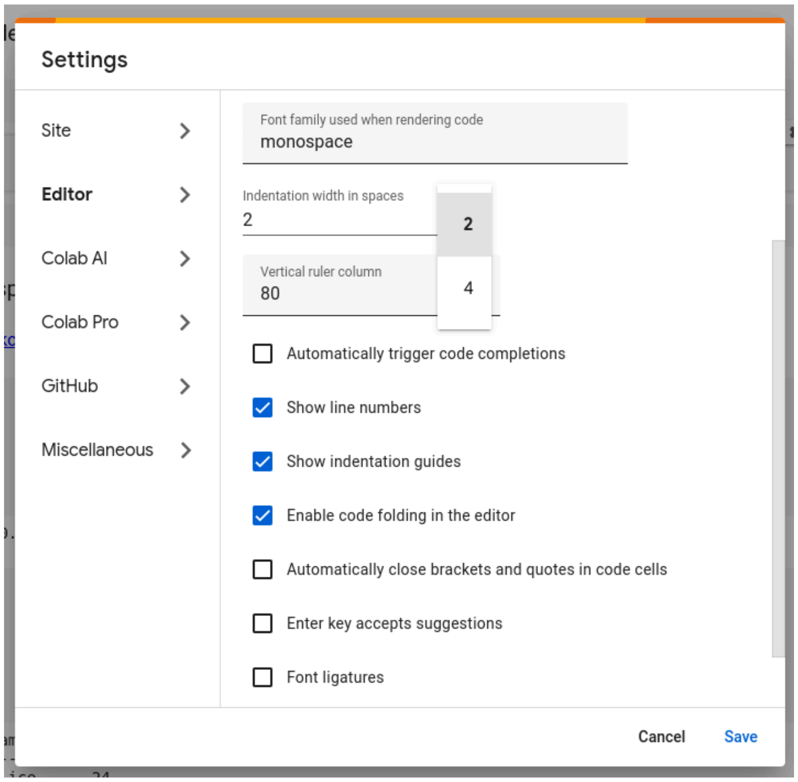
1. in code cell, click cog wheel in cells upper right toolbar.



```
# this is a code cell

```

2. update editor settings "**indentation width in spaces**" with drop-down selection. sensei uses "2".
3. update editor settings "**show line numbers**" as selected - necessary for common reference during remote debugging sessions.
4. update editor settings "**show indentation guides**" as selected - bc python is unforgiving about poor indentation.
5. update editor settings "**enable code folding in editor**" as selected - bc its tidier, less distracting wrt finished portions of code.
6. save, obviously!



Settings

Font family used when rendering code: monospace

Indentation width in spaces: 2 (options: 2, 4)

Vertical ruler column: 80

- [ ] Automatically trigger code completions
- [✓] Show line numbers
- [✓] Show indentation guides
- [✓] Enable code folding in the editor
- [ ] Automatically close brackets and quotes in code cells
- [ ] Enter key accepts suggestions
- [ ] Font ligatures

2. indentation is life

python relies on indentation to know what goes together. it also has expectations about that.

```
# example 01: this is fine
print("first")
print("second")
print("third")

# example 02: this is not fine
print("fourth")
  print("fifth")
print("sixth")

```

Indentation is **crucial** in Python, unlike many other languages, because it defines code blocks. Here's the main idea:

- Spaces or tabs (choose one consistently) create indentation. - The level of indentation determines which code belongs together.

Think of indentation like levels in a building. The ground floor is the main program. When you enter a room (like an `if` statement), you indent the code inside (deeper level). To "exit" the room, you un-indent (go back to the previous level).

Key points:

- Consistent indentation (same number of spaces for each level) is **essential**.
- Mixing spaces and tabs can lead to errors. ~so dont do it!!
- Indentation defines the body of statements for:
  - `if`, `elif`, `else` statements
  - `for` and `while` loops
  - Functions

Example:

```
if number > 0:
    print("The number is positive.")  # This line is indented, belonging to the "if" block
else:
    print("The number is non-positive.")  # This line is also indented, belonging to the "else" block

for i in range(5):
    print(i)  # This line is indented, belonging to the "for" loop body
```

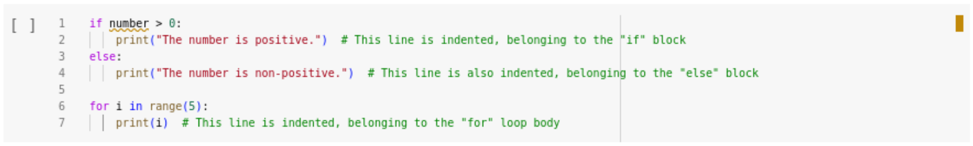
Remember: Indentation is like invisible curly braces in Python. Use it carefully to structure your code clearly and avoid errors!

In Colab code cells, red and yellow underlining help identify potential errors or issues in your code.

- **Red underlines** typically indicate syntax errors or typos. These are mistakes that prevent the code from running correctly.
- **Yellow underlines** might signify potential issues or warnings. They don't necessarily prevent the code from running but could suggest areas for improvement or unexpected behavior.

These underlining are like spellcheck for your code, helping you catch mistakes before you run the code.

Example:



```
[ ]  1  if number > 0:
     2      print("The number is positive.")  # This line is indented, belonging to the "if" block
     3  else:
     4      print("The number is non-positive.")  # This line is also indented, belonging to the "else" block
     5
     6  for i in range(5):
     7      print(i)  # This line is indented, belonging to the "for" loop body
```

on line 01 (**make sure your line numbers are turned on!**), the yellow squiggly lines are a warning that something might be wrong. lines 02, 04, 07 have vertical bars that indicate significant indentations. (**make sure your indentation guides are turned on!**)

3. data types

immediately useful native data types: int, float, complex, str, list, boolean.

```
a = 1   # python decides a is int(eger)
print(f"a = {a}, {type(a)}. ~see? i told you so")

b = 0.2 # python decides b is float
print(f"b = {b}, {type(b)}")

```

```
a = 1, <class 'int'>. ~see? i told you so
b = 0.2, <class 'float'>
```

wait, you interrupt. thats a fancy print statement. (will there be a quiz? does my life depend on that?) so what, i reply. (no. and maybe?) youll see different ways to use "print()" in this wee fyi. its not like this notebook will vanish. save a copy; reference when you feel like it.

```
c = 3 + 4j # python decides c is complex
print(f"c = {c}, {type(c)}")

```

```
c = (3+4j), <class 'complex'>
```

wait, you say. why is it "j"? bc during one skirmish in wars of various naming conventions, electrical engineers (who use *i* for electrical current) smacked down the math guys. so there.

```
d = "la la la"
print(f"d = {d}, {type(d)}")

f = True
print(f"f = {f}, {type(f)}")

```

```
d = la la la, <class 'str'>
f = True, <class 'bool'>
```

these below are booleans too.

```
f2 = 1==2
f3 = 1>2
print(f"f2: {f2}")
print(f"f3: {f3}")
print(f"1<2: {1<2}")
print(f"1!=2: {1!=2}") # not equal
print(f"1<=2: {1<=2}")

```

```
f2: False
f3: False
1<2: True
1!=2: True
1<=2: True
```

```
e = []
e2 = [1,2,3]
e3 = [4,"5",0.6]
e4 = [7,[8,9]]

print(f"e: {e}")
print(f"e2: {e2}")
print(f"e3: {e3}")
print(f"e4: {e4}")

```

```
e: []
e2: [1, 2, 3]
e3: [4, '5', 0.6]
e4: [7, [8, 9]]
```

yes, you can mix it up like that. numpy.array is strict (and therefore more reliable in content) but python list is not.

matrices

lists ~ arrays.

as an alternative, numpy.array is highly recommended.

```
e5 = [[1,2],[3,4]]
print (f"e5: {e5}")

e6 = e5
print (f"e6: {e6}")
print (f"e6[0][0]: {e6[0][0]}")

e6[0][0] = 2e2
print (f"e6[0,0]: {e6[0][0]}")
print (f"e6: {e6}")

print()
print (f"e5: {e5} ~ eine minute, bitte...")

```

```
e5: [[1, 2], [3, 4]]
e6: [[1, 2], [3, 4]]
e6[0][0]: 1
e6[0,0]: 200.0
e6: [[200.0, 2], [3, 4]]

e5: [[200.0, 2], [3, 4]] ~ eine minute, bitte...
```

wait, what? thats bc more complicated data structures need this:

```
import copy

e5 = [[1,2],[3,4]]
print (f"e5: {e5}")

e6 = copy.deepcopy(e5)
e6[0][0] = 2e2
print (f"e6: {e6}")
print (f"e5: {e5}")

```

```
e5: [[1, 2], [3, 4]]
e6: [[200.0, 2], [3, 4]]
e5: [[1, 2], [3, 4]]
```

yes, this is annoying.

also, check out numpy.copy() and numpy.ndarray.copy()

wrt referencing indexes: lists use a[i][j][k], numpy arrays can optionally use a[i,j,k]. more as it appears during semester, within relevant lecture notes.

4. operators

```
g = []
g.append(1 + 2)
g.append(1 - 2)
g.append(1*2)
g.append(1/2)
g.append(2*2)
g.append(pow(2,2))

print(g)
```

```
[3, -1, 2, 0.5, 4, 4]
```

```
import math

g = math.sqrt(2)

print(g)
```

```
1.4142135623730951
```

```
import numpy as np

r = 3
a = np.pi*pow(r,2)
```

```
print(f"area of disk of radious {r}: {a}'
```

```
area of disk of radious 3: 28.274333882308138
```

5. control flow

if statements. note the indentations and colons.

```
a = 1
b = 2

if a==b:
  print("a = b, hooray")
elif a!=b:
  print("a <> b, alas")
  if a>b:
    print("and a > b")
  elif a==3:
    print("a = 3")
  elif b==3:
    print("b = 3")
  else:
    print("this looper is tired and needs a nap")
    if a<b:
      print("oh, and a < b happened, too")
else:
  print("you can see, right? that this statement is never reached")

```

```
a <> b, alas
this looper is tired and needs a nap
oh, and a < b happened, too
```

for-loops.

```
for i in range(3):
  print(i)

print() # spacer
for i in reversed(range(3)):
  print(i)
```

```
0
1
2

2
1
0
```

python range has three arguments: start,stop (before), step. if one arugment used, it is stop; if two arguments used, they are start, stop.

- range()
- reversed()

list comprehension, a slick option for sometimes.

- @python.org

```
bs = ["b1",2,"b3"]
c = [b for b in bs]
print(f"list comprehension, plain jane: {c}")

c = [b for b in bs if type(b)==int]
print(f"list comprehension, a little picky: {c}")
```

```
list comprehension, plain jane: ['b1', 2, 'b3']
list comprehension, a little picky: [2]
```

6. display, print, simple tabular

@stackoverflow

```
# https://docs.python.org/3/library/pprint.html

import pprint

pp = pprint.PrettyPrinter(depth=2)
pp.pprint([0.1])

```

```
[0.1]
```

```
# https://pypi.python.org/pypi/tabulate

from tabulate import tabulate

print(tabulate([['Alice', 24], ['Bob', 19]], headers=['Name', 'Age']))
print()
print(tabulate([['Alice', 24], ['Bob', 19]], headers=['Name', 'Age'], tablefmt='orgtbl'))

```

```
Name      Age
------  -----
Alice      24
Bob        19

| Name   |   Age |
|--------+-------|
| Alice  |    24 |
| Bob    |    19 |
```

```python
# https://pypi.python.org/pypi/PrettyTable

from prettytable import PrettyTable

t = PrettyTable(['Name', 'Age'])
t.add_row(['Alice', 24])
t.add_row(['Bob', 19])
print(t)

```

```
+-------+-----+
|  Name | Age |
+-------+-----+
| Alice |  24 |
|  Bob  |  19 |
+-------+-----+
```

7. plots

do the (extra credit) colab tutorial.