Game Programming with Python

Zoom

- Open a browser to zoom.us
- Click on `Join` at the top
- Enter 388 023 5098
- Launch meeting in browser
- Don't bother with audio ("Continue")
- Watch my show

Introduction

- Some of this workshop material is based on Invent Your Own Computer Games with Python (4e) by Al Sweigart (free online).
- Download the code along notebook for this class to your PC from here: `tinyurl.com/trio-games-colab` and save it as `trio.ipnyb` .
- The notebooks were rendered from my own Emacs Org-mode notebooks - they are also available online at tinyurl.com/trio-games-org

Overview

1. Programming as an art (with or without AI).
2. Programming with IPython, IDLE, and Google Colaboratory.
3. Programming with Python: numbers, operations, variables.

4. Simple programs: “Hello World” and user input/output.
5. Quiz 1.
6. Python loops and conditional statements.
7. Random numbers.
8. Write and understand a simple game.
9. Quiz 2.
10. What do to next.

Why should you (still) learn how to program?

./img/programming_is_fun.webp

- Programming is fun.
- Programming is a useful skill to have.
- Programming trains your brain to think logically.
- Programming gets you used to making and fixing mistakes.

Will AI not do all the programming for us?

“So much of what we do as software developers is just magic to people. It’s really hard for people outside of software to gauge how complicated some tasks are, or what software is capable and not capable of doing. So AI just seems magical, but I have a really dim and critical view of ChatGPT and a lot of these large language models and various tools. It’s not like blockchain or NFTs. There is something there, but there is also just so much hype surrounding these tools. So what I have to tell people is, yes, it absolutely is worth it to learn how to program.”

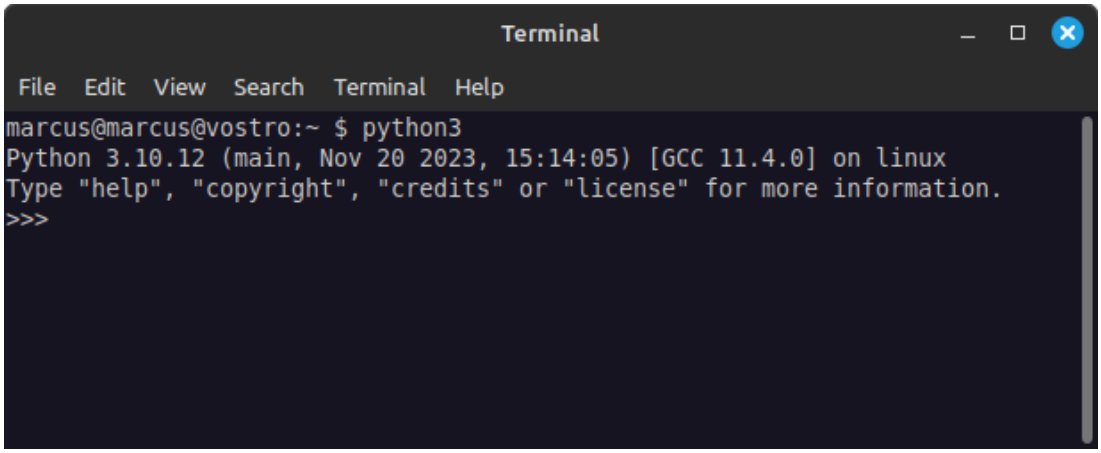
Al Sweigart (Stackoverflow.com podcast, May 24, 2024)

What does this cartoon mean to you?

- CS = Computer Science
- GIS = Geographical Information System (like Google Maps).

Getting started with IPython

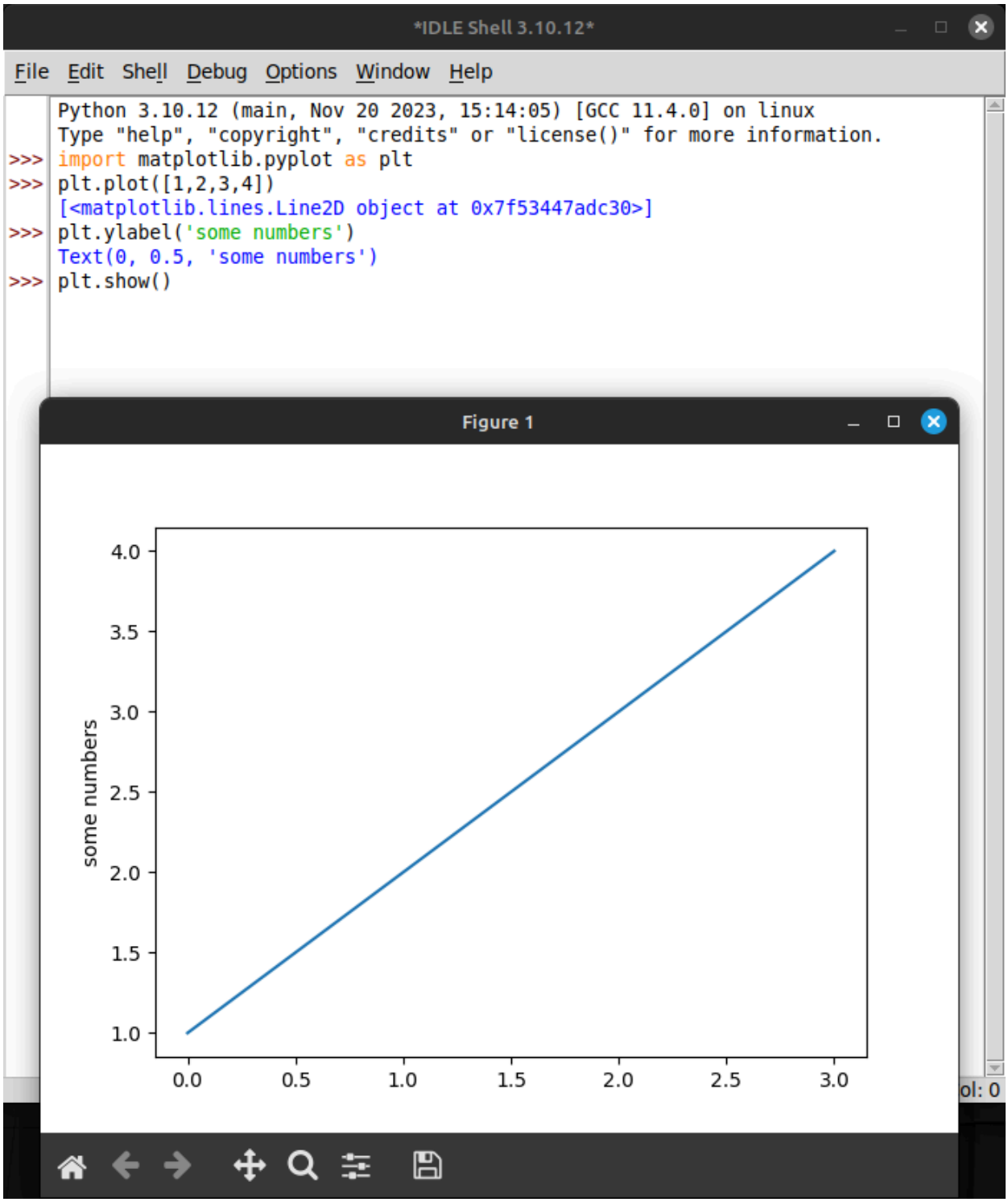
- IPython is a Python *shell* program - an environment to connect to the Python interpreter. It's not the same as
- The Python interpreter takes your commands and executes them directly.
- Alternatively, you must download and install the Python interpreter on your personal computer (from python.org).
- If you have Python on your PC, you can open a command line (CMD) and open the interpreter (aka console or shell) with the command `python3`:

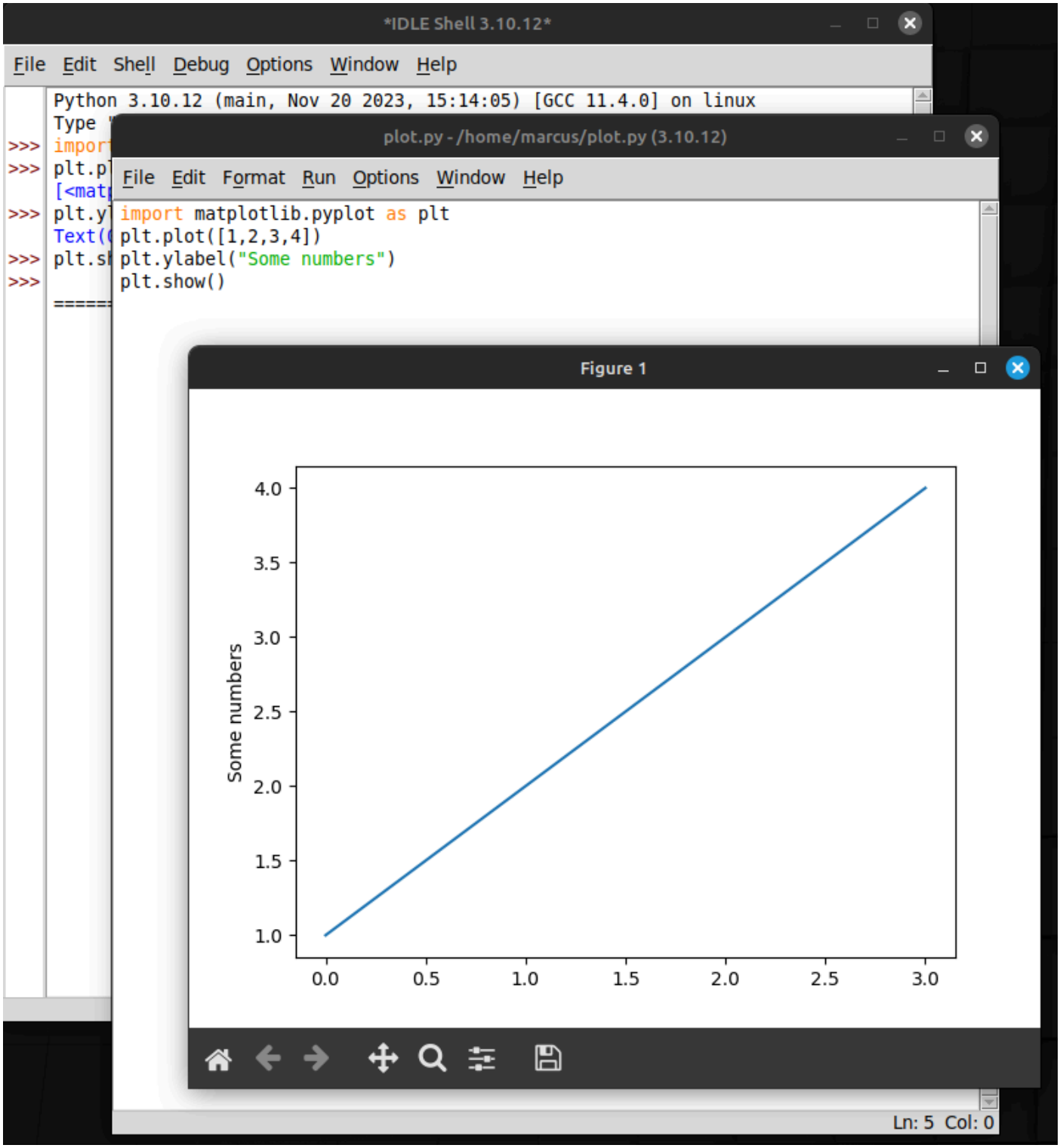


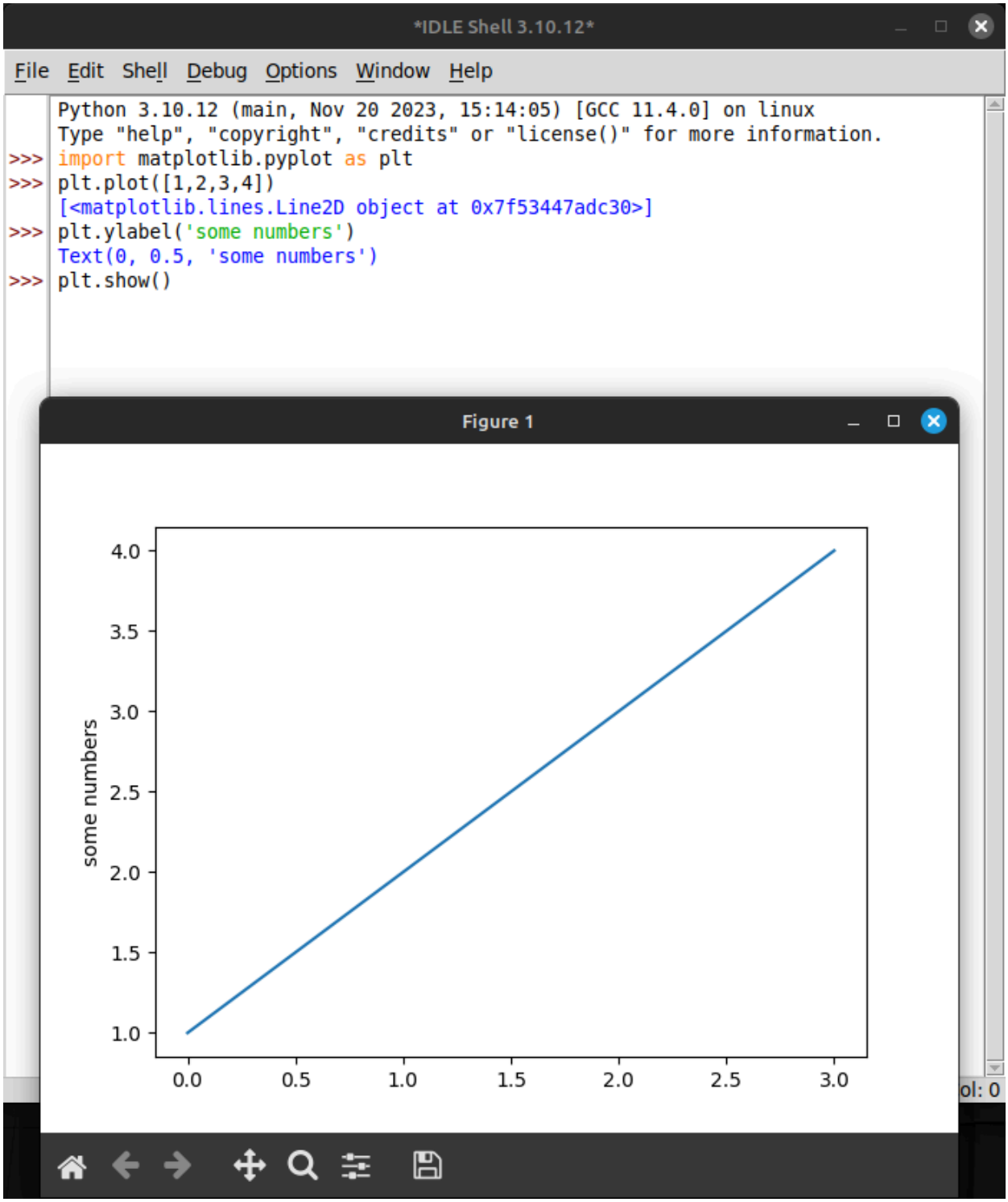
- The '3' suggests that there was a 'python2': on many computers, entering `python` will start Python 2 instead of Python 3.
- Confusingly, Python 2 and Python 3 are not compatible.

Getting started with IDLE

- When you have Python, you also have another program called IDLE, which can be started from the command line with `idle`.
- IDLE (Interactive DeveLopment Environment) is like a word processor for writing Python programs.
- It's a separate window where you can enter commands and execute them.




IDLE Shell 3.10.12
File Edit Shell Debug Options Window Help
Python 3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0] on linux
Type
>>> import
>>> plt.p
[<mat
>>> plt.y
Text(
>>> plt.sh
>>>
plot.py - /home/marcus/plot.py (3.10.12)
File Edit Format Run Options Window Help
import matplotlib.pyplot as plt
plt.plot([1,2,3,4])
plt.ylabel("Some numbers")
plt.show()
Figure 1
Some numbers
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
Ln: 5 Col: 0



- IDLE is a so-called "Integrated Development Environment" (IDE) that allows you to perform different operations during program development on one platform: writing, debugging and executing code.

Getting started with Google Colaboratory

- Fortunately, Google offers a free IPython application called Colaboratory, which we will use to do all of our coding.
- To open, you must have a Google account. In a browser (any will do but Google Chrome works best), open `colab.research.google.com`.
- In this notebook, you can add text and code, and you can run the code.

- To experiment with that, open a new notebook from the `File` menu and code alongside me.
- The text can be formatted using so-called Markdown language:
  - `#` will create a headline and a section
  - `##` will create a headline and a subsection
  - `` ` ` `` will format text as code
  - `![img](URL)` will load an image from the address URL
  - etc.
- `CTRL + ENTER` will execute a code cell, and `SHIFT + ENTER` will execute it and create a new code cell.
- Examples:
  i. Create a text cell with the headline “My first text cell”
  ii. Create another text cell with the sub-headline “My first code cell”
  iii. Create a code cell
  iv. In the code cell, type this code & run it with SHIFT + ENTER

```
import matplotlib.pyplot as plt
```

  v. In the next code cell, type this code & run it with CTRL + ENTER:

```
plt.plot([1,2,3,4])
plt.ylabel('some numbers')
plt.show()
```

- You now have all the ingredients of an interactive data science notebook: text, code, and output.
- Give the notebook a title, e.g. “Colabdemo.ipynb” and save it. This file will now automatically be saved to your Google Drive account.

Getting started with Python

- At the top of the welcome screen, you find the `File` menu: open it and choose `Upload notebook`, then browse your PC to upload the file `trio.ipynb` that you downloaded at the start.
- The notebook contains all the text from my own notebook, with little exercises and space for you to code along.

Manipulating values

Manipulating integer values

We’ll start by learning how to manipulate numbers (‘arithmetic’).

- In the code block, execute the operation `2 + 2` (`CTRL + ENTER`).

```
2 + 2
```

- In the next code block, write `2 + 2` on one line, and `2 - 2` on the next line, then execute the block:

```
2 + 2
2 - 2
```

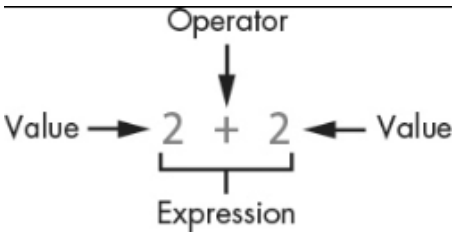
- [In Colab] Where did the first result go? Answer: you need to use `print` for every expression that you want to print out, otherwise only the last one evaluated will be shown.

Using operators, floats, integers

- You can add, subtract, multiply and divide in Python.

```
print(1e3 * 1e-3)
print(1/1)
```

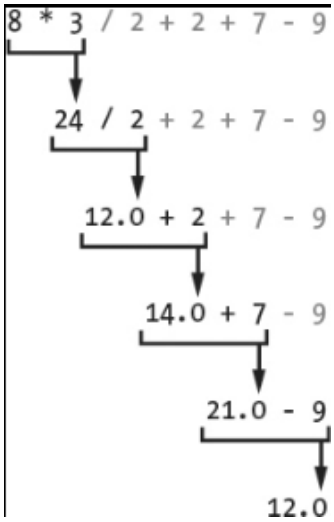
- The first line above uses scientific notation for large numbers: `1e3` is `10 * 10 * 10 = 1000`, and `1e-3` is `1 / (10 * 10 * 10) = 1/1000` or `0.001`.
- Both operations result in a decimal (or floating-point) number (`1.0`), or *float*, rather than a whole (or integer) number (`1`).
- A number like `1` or `2.0` is a *value*. A math problem like `2 + 2` is an *expression*. Expressions are made up of values connected by operators (`+`).



- The computer is obsessed with evaluating expressions. In the next code block, evaluate some expressions and `print` the results:
  i. `2 + 2 + 2 + 2 + 2`
  ii. `8*6`
  iii. `10 - 5 + 6`
  iv. `2 + 2`

```
print(2 + 2 + 2 + 2 + 2)
print(8*6)
print(10 - 5 + 6)
print(2 +          2)
```

- When an expression is evaluated, Python has to observe an order of operations (“P+E+MD+AS”). The expression is always evaluated to a single value:



- Run the first line of the code in a code block and `print` the result:

```
print(8 * 3 / 2 + 2 + 7 - 9)
```

- Test question: Will the following expressions give the same or different results?

```
print(8 * 3 / 2 + 2 + 7 - 9)
print(2 + 7 - 9 + 8 * 3 / 2)
print(2 + 7 - 9 + 8 * (3 / 2))
print(2 + 8 * (3 / 2) + 7 - 9)
```

Making syntax errors

- Entering ` 5 + ` generates a `SyntaxError` because the `+` operator is binary and requires two arguments on either side:

```
5 +
```

- Syntax errors result from not observing the rules of the language - it's as if Yoda was saying "Home I go". This violates the SPO rule of English syntax - Subject + Predicate + Object.
- The difference between humans and machines: we can often, the computer can never recover from syntax errors.

Storing values in variables

- A variable is like a box that can hold a value.
- In the next code block, store the integer number `15` in a variable called `spam`[fn:1].

```
spam = 15
```

- You've just written a *statement* or more specifically an *assignment statement* using the assignment operator `=`. There's no output until you ask for the value stored in `spam`.

```
print(spam)
```

- Python is case-sensitive, i.e. `SPAM` is different from `spam` or from `Spam`. You can test that by printing all of these:

```
print(spam)
print(SPAM)
print(Spam)
```

- The last two attempts result in a `NameError` because these variables were `not defined`, i.e. they were never assigned values.

Computing with variables

- Once a variable is defined, you can use it to compute. In the next code block, `print` the expressions `spam + 5` and `spam * spam`:

```
print(spam + 5)
print(spam * spam)
```

- In fact, you don't need two lines for this: put both expressions in the same `print` command:

```
print(spam + 5, spam * spam)
```

- Now change the value of `spam` to `3` and print the expressions again:

```
spam = 3
print(spam + 5, spam * spam)
```

- Do you think it's possible to do all of that in the `print` command, like this:

```
print(spam = 3, spam + 5, spam * spam)
```

- You encounter a third kind of error, a `TypeError`: inside `print`, `spam` is not recognized as part of `spam = 3`.
- However, if you change the `=` in the last command to a `==`, the code works:

```
print(spam == 3, spam + 5, spam * spam)
```

- This is because now you're printing a *value* as required by Python, the value is `True` because `spam` is actually equal to `3`. The `==` is a relational operator. It tests the equality of its left and its right hand operand.
- In the next code block, first alter the value of `spam` by adding `2` to itself like this: `spam = spam + 2`. In the following line, repeat the previous `print` command:

```
spam = spam + 2
print(spam == 3, spam + 5, spam * spam)
```

- Now, `spam == 3` is `False`, because the new value is 3 + 2 = 5.
- In the next code block, define two more variables, `bacon` with the value `10`, and `eggs` with the value `15`.

```
bacon = 10
eggs = 15
```

- Enter `spam = bacon + eggs` in the next code block, then check the value of `spam`:

```
spam = bacon + eggs
print(spam)
```

Summary I

- Expressions are values like `2` or `5.0` combined with operators like `+` or `/`.
- Expressions are evaluated and reduced to a single value.
- Values can be stored in variables to be remembered and used later.
- Python errors include `SyntaxError`, `TypeError` and `NameError`.

Writing programs

Using string values

- In Python, text values are called *strings*. They can be used just like integer or float values, and you can store them in variables.
- Python recognizes text values when they are enclosed in (single or double) quotation marks: `'"spam"'` is a string, `spam` is a variable.
- Store the string `hello` in the variable `spam`, then `print` it:

```
spam = 'hello'
print(spam)
```

- Strings can have any keyboard character in them and they can be as long as you like:

```
print("ijdinfnns d    \n   ***&&^6///34/$$$\n\
once upon the time in a galaxy far, far away..."x)
```

- In the last example, Python recognized two special characters: `\n` (new line) and `` (continue here).
- Strings can be *concatenated*: enter `'Hello' + 'World!'` in the next code block:

```
print('Hello'+'World!')
```

- What's behind this? All values have a *data type* (integer, float, string, or Boolean), and the `+` operator works differently on them:

```
print(2 + 2) # add two integers
print(2. + 2.) # add to floats
print('2' + '2') # add two strings
print(True + False) # add two Booleans (True is 1 and False is 0)
```

Creating the Hello World program

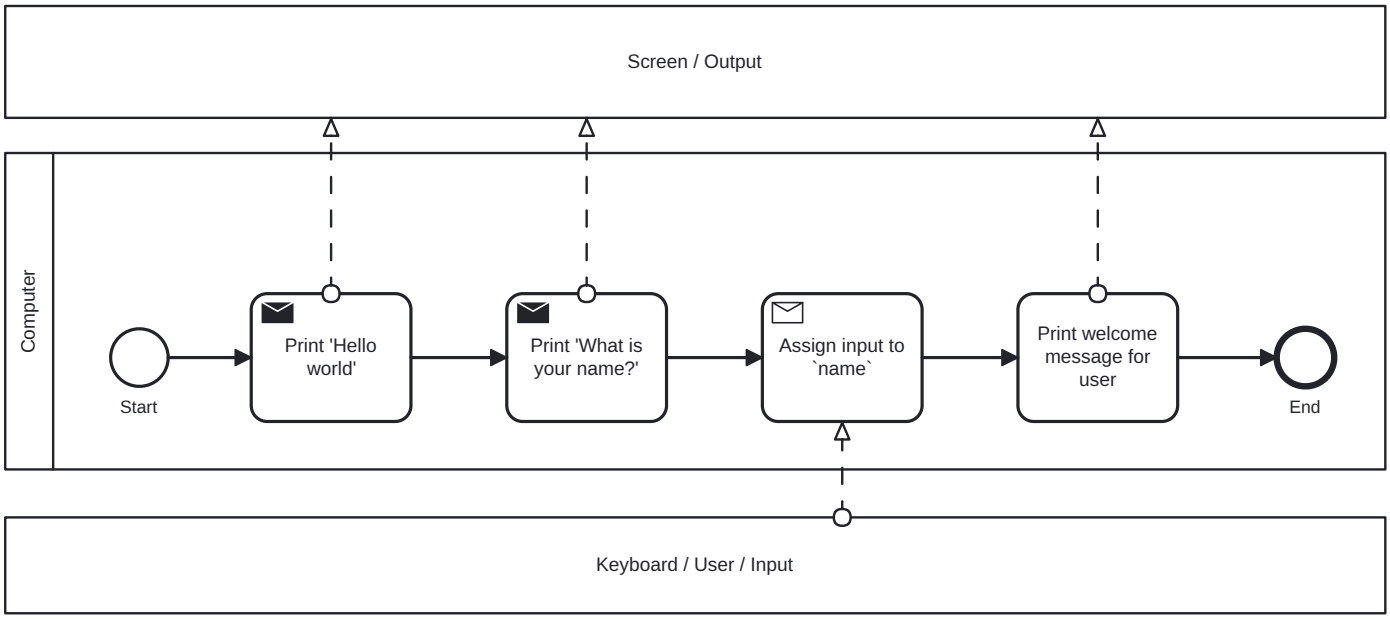
- It's traditional for programmers to make their first program display `Hello world!` on the screen[fn:2]
- Enter the following code in the code block:
  i. `# This program says hello and asks for my name`
  ii. `print('Hello world!')`
  iii. `print('What is your name?')
  iv. `name = input()`
  v. `print('It is good to meet you, ' + name)

```
# This program says hello and asks for my name`
print('Hello world!')
print('What is your name?')
name = input()
print('It is good to meet you, ' + name)
```

- Let's analyze the program:
  i. This is a comment - Python ignores everything after `#`
  ii. Displays a string on the screen - the program title.
  iii. Displays a string on the screen - a question for the user.
  iv. Asks for keyboard input and assigns it to the variable `name`.
  v. Concatenates the welcome and the value `name` of name & prints them.
- When you write your own programs, it is useful to add this information to the code using comments:

```
# This program says hello and asks for my name`
print('Hello world!')  # Print title
print('What is your name?')  # Ask user for input
name = input() # assign keyboard input to `name`
print('It is good to meet you, ' + name) # print result as concatenation
```

- In practice, you would first write the comments as a form of pseudocode, and/or put them in a process model. The more complicated a program is, and the more people are working on it, the more important it is that you follow these development practices!
- BPMN Process model for the Hello world program (created at bpmn.io):



Getting input, printing output

- `print()` and `input()` are built-in functions.
- The value between the parentheses of a function is called its *argument*, like for mathematical functions .
- When the function is called, the argument is passed to the function for evaluation.
- Examples:

```
# Print the string argument "hello world"
print("hello world")

# Print the number argument 2
print(2)

# Print the value of the expression 2 + 2
print(2+2)
```

- You can get short help on any function (or keyword) with the `help` function. In the next code block, pass the name of the `print` function as an argument to the `help` function:

```
help(print)
```

- Do the same thing for `input`: get `help` using the `help` function

```
help(input)
```

- Both `help` texts contain a lot of technical information that you may not understand (yet). Especially when you encounter a new function, it's worth going down the rabbit hole of documentation to understand absolutely everything that the `help` can tell you.
- Here is the `input` command from the program again:

```
name = input()
print('Hello, ' + name)
```

- What happens here? The function `input` is called without an argument. As the `help` explains, it reads "a string from standard input". Standard input (*stdin*) in this case means the keyboard.
- Standard input could also be passed to a Python script: after tangling the single command above as a Python file `input.py`, it can be run on the command line if `input` is a file containing input:

```
echo 'Marcus' > inputFile
python3 input.py < inputFile
```

Forgetting and naming variables

- What happens to the variables when the program is finished?
- It depends:
  i. If you're working in an interactive notebook like an IPython shell, or in Emacs Org-mode, the variables are alive as long as the notebook session is running.
  ii. If you run a program on the command line like `python3 input.py` above, everything is gone when the program is finished.
- Your variables have to be named by you. There are a few rules and recommendations for that:
  i. Don't start a name with anything but a (lowercase) letter (underscores are reserved, numbers or operators are not allowed)
  ii. Observe the fact that variable names are case-sensitive: `SPAM` is not the same as `spam`.
  iii. You must not have whitespace (empty characters) within the name.
  iv. Variable names are usual lower case. You can form longer names either by connecting them with underscore `_` or with *camelCase*: for example: `my_number` or `myName`.

Summary II

- All values have a data type (float, integer, string, or Boolean).
- Strings must be enclosed in single or double quotation marks.
- Strings can be concatenated with the `+` operator.
- Functions carry out complicated instructions, they are called with or without arguments, e.g. `print(2)` or `input()`.
- Functions can be used anywhere a value is used: `name=input()`.

[[https://quizizz.com/admin/quiz/6661f06fc8188d75936738cd/python-basics-quiz?fromSearch=true&source=][Quiz 1: Python Basics

]]

Writing a game program

Defining the game

- We're going to bring the last few topics together in a complete little game script, a Guess the Number game.
- In this game, the computer will think of a secret number from 1 to 20 and ask the user to guess it. After each guess, the computer will tell the user whether the number is too high or too low. The user wins if they can guess the number within six tries.
- The game uses many new Python tools:
  i. random numbers
  ii. repeating code chunks
  iii. grouping and indenting code
  iv. selecting choices based on conditions
  v. converting values to different data types
  vi. breaking out of loops

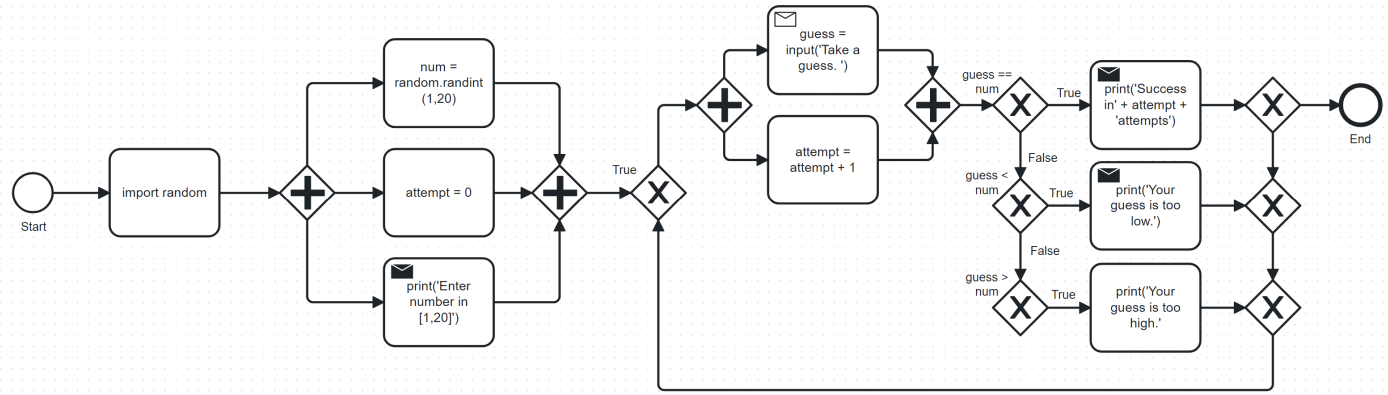
Planning the game

- This example also demonstrates an exemplary solution path:
  i. Understand what's asked from you (**requirements**)
  ii. Understand what the program needs from you (**input**)
  iii. Understand what's the result supposed to look like (**output**)
  iv. Plan the process without syntax (**pseudocode**)
  v. Create a process **diagram** (with commands)
  vi. Code the Python program (**source code**)
  vii. Run, test and debug the source code (**production code**)
  viii. Fix pseudocode/diagram accordingly (**feedback**)
  ix. Identify **extensions** (other things you might like)
  x. Implement extensions (repeat steps 4-8).
- When you run the program, the output should look like this:

```
Enter number between 1 and 20:
Take a guess: 10
Your guess is too high.
Take a guess: 2
Your guess is too low.
Take a guess: 8
Your guess is too high.
Take a guess: 3
Your guess is too low.
Take a guess: 7
Good job! You guessed my number in 5 guesses!
```

- The program should generate a random number between 1 and 20.

- Enter the source code into the IDLE file editor, or into Colab, and save as `guessTheNumber.py`.
- Solution path/pseudocode (code highlighted)
  i. `import random` module.
  ii. Generate a (secret) `random` number.
  iii. Store number in variable `num`.
  iv. Set `attempt` counter (number of guesses) to `0`.
  v. Get `input` number `guess` from user.
  vi. Increase `attempt` by 1
  vii. Check if `guess` is the same as `num`
  viii. `print` success message and `attempt` value
  ix. End program
  x. Otherwise, check if `guess` is smaller than `num`
  xi. `print` information
  xii. Otherwise, check if `guess` is larger than `num`
  xiii. `print` information
  xiv. Return to step 3
- The BPMN Process diagram is fairly complicated compared to the previous example:



- Solution Python code (16 + 5 lines):

```
# import random module
import random
# pick random number between 1 and 20
num = random.randint(1,20)
# set attempts counter to 0
attempt = 0
# ask user for number guess
print('Enter number between 1 and 20: ')
# infinite loop until number is guessed
while True:
    guess = int(input('Take a guess: '))
    attempt = attempt + 1
    if guess < num:
        print('Your guess is too low.')
        continue
    elif guess > num:
        print('Your guess is too high.')
        continue
    else:
```

```
        print('Good job! You guessed my number in ' + str(attempt) + ' guesses!')
        break
```

Generating random numbers

- To generate a secret guess, we use a (pseudo-) random number generator. In Python, such a generator is contained in the `random` package.
- To make this package available, you need to *install* it to your computer and then *load* it in the Python session where you need it.
- In Colab, the package is already installed and only has to be loaded:

```
import random
```

- We can now pick a random number between 1 and 20 using the `randint` function in `random`: run the following code multiple times to see how it works.

```
print(random.randint(1,20))
```

- The `.` operator accesses the `random` package. You remember that we loaded a graphics package, `matplotlib.pyplot` earlier, and gave it an alias, `plt`[fn:3]:

```
import matplotlib.pyplot as plt
```

- To access the `plot` function in the package, we called the function twice: first to pass four points for plotting, and then to make the plot appear on the screen:

```
plt.plot([1,2,3,4])
plt.show()
```

- You need randomness in many games - even board games use dice, and many game actions, e.g. by NPCs, are randomized.

Repeating code

- The next part of the code that may be new to you if you never programmed before is the line `while True:`
- This is an infinite loop: the `while` command enters the loop followed by a test. The generic form of the command is:

```
while [test]:
    # do something
```

- The result of the test is either `True` in which case the loop is entered, or `False`, in which case it is left again without doing anything.

- Let's look at a few examples:

```
i = 1
while i < 3:
    i = i + 1
    print(i)
```

- Let's analyze:
  i. Here, `i` is set to 1. When the `while` is encountered, `i < 5` is tested. Since it's `True`, the statements in the loop body are run: `i` is increased to 2, and printed.
  ii. The loop is entered a second time: the test `2 < 5` is still `True`, `i` is increased to 3, and printed.
  iii. The loop is entered a third time: the test `3 < 5` fails - it is evaluates to `False`, and the loop commands are not executed.
- As a challenge, change the `while` loop so that it starts at `i = 5` and tests if `i > 0`, so that the output is: `4 3 2 1 0`.

```
i = 5
while i > 0:
    i = i - 1
    print(i)
```

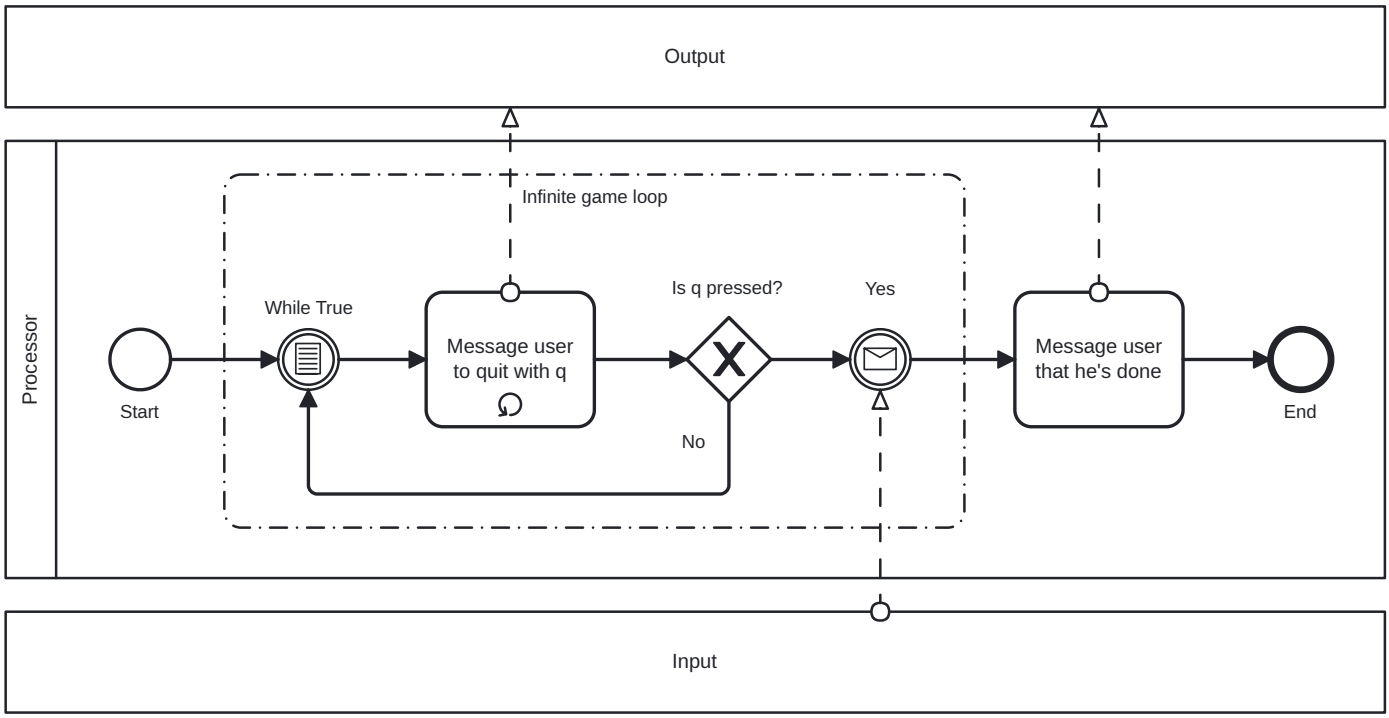
- Coming back to our game: If the test reads `True` then the condition *never* fails and the loop will keep running forever!
- To stop the game inside an infinite loop, we must take extra measures: we must `break` out of the loop.
- Here is an example: This loop runs exactly once and then exits because of the `break` command.

```
while True:
    print("Infinite loop!")
    break
print("Done!")
```

- The next one runs until `q` is entered. It prints the message to the screen and then halts waiting for input:

```
while True:
    print("Infinite loop...until you type q")
    if input()=='q': break
    print("Done!")
```

- The last example checks a condition after the `if` keyword: this is called a *conditional statement*. Seen through process model eyes, this last code chunk looks like this:



Checking conditions

- The core of the infinite game loop also has a conditional statement. Instead of one check, it has two: namely, if the user's guess, stored in `guess`, is greater or smaller than the computer's (secret) number:

```
if guess < num:
    print('Your guess is too low.')
    continue
elif guess > num:
    print('Your guess is too high.')
    continue
else:
    print('Good job! You guessed my number in ' + str(attempt) + ' guesses!')
    break
```

This is what happens inside the loop:

1. If the guess is smaller than the computer's number, the user is told that it is, and we `continue` with another guess.
2. If the guess is greater than the computer's number, the user is told that it is, and we `continue` with another guess.
3. If the guess is neither smaller nor greater than the computer's number, we must have guessed it: then we print the result and `break` out of the loop to finish.

Getting the user's number

- At the start of the loop, we get the user's guess and store it in the variable `guess`.
- We get this number from the keyboard with `input`:

```
guess = int(input())
```

- You notice that we did not write `guess = input()`. Why? Let's see:

```
print('Enter a number between 1 and 20:')
guess = input()
print(guess < 20)
```

- When you run this code, you get a `TypeError`:

```
TypeError: '<' not supported between instances of 'str' and 'int'
```

- The problem is that you cannot compare a string (`str`) and an integer (`int`) - and with `input` you can only import strings from the keyboard.
- To fix this, you must *convert* the string to an integer. This only works, of course, if the converted string can be recognized as a number: it works for `"2"` but not for `"a"` or `"2 + 2"`:

```
print(int("2"))
print(int("2 + 2"))
print(int("a"))
```

- So `guess` holds not the string value of the user's number but the integer value, which can be compared with the computer's number.

Printing the result

- Fortunately, we have already understood the concept of conversion: in the printout of the result, another conversion takes place, but this time the other way around, from integer to string:

```
print('Good job! You guessed my number in ' + str(attempt) + ' guesses!')
```

- In the case of `str`, any number can be turned into a string:

```
print(str(1e+3))
print(str(0.001))
```

- One way of testing if a value is a string or a number is by concatenating it with another string:

```
print("The number is " + str(0.001))
print("The number is " + 0.001)
```

- As before, the second command fails with a `TypeError`:

```
TypeError: can only concatenate str (not "float") to str
```

Putting it all together

- In the next code block, let's assemble the whole program and run it:

```
import random  # import random module
num = random.randint(1,20) # pick random number in (1,20) - computer's number
attempt = 0 # initialize number of attempts
print('Enter number between 1 and 20:') # ask for user guess
while True:  # start the infinite loop
    guess = int(input('Take a guess: ')) # Prompt user input & store in guess
    attempt = attempt + 1 # increase number of attempts
    if guess < num: print("Too low")
    elif guess > num: print("Too high")
    else:
        print("Attempts: " + str(attempt))
        break
```

- You find yet another solution in the textbook on page 22.

Program extensions and lessons learnt

- Program extensions:
  i. Make program safe against no/wrong input (exception handling): currently, it terminates with an error if a floating-point number or a letter or nothing is entered by mistake.
  ii. Exchange the infinite `while` loop by a `for` loop with a set number of allowed guesses (most games don't go on forever).
- What's important to remember:
  i. For best productivity and learning, follow a solution path - don't just "code away"
  ii. For best learning effects find different solutions to the same problem.
  iii. For best results, handle exceptions. Balance exception handling with usability and performance.
  iv. There is always more than one solution, usually many. There is no best solution to a programming problem that satisfies all requirements, even the unspoken ones, equally well.

Summary III

- Expressions as part of an `if` or `while` statement are conditions. They evaluate to Boolean (truth) values.
- `break` and `continue` are flow control statements to break out of a loop or go back to the start of the loop.
- `print` and `input` serve the standard output (stdout) and the standard input (stdin) data stream, or output (e.g. to the screen) and input (e.g. from the keyboard).
- `int` and `str` are functions that convert strings and numbers into integers and strings, respectively.

Quiz 2: Python Programming

What to do next

- You've just completed the first three chapters of this book: "Invent Your Own Computer Games with Python, 4th ed." by Al Sweigart
- The completed notebook for the course is available at tinyurl.com/trio-games-colab-solution
- The whole book is (legally) freely available online: inventwithpython.com
- The book is the basis of a COR 100/Year One college course "Game Programming with Python".
- These books by the same author are great (not only) for beginners:
  i. Automate the Boring Stuff with Python (2e, 2019)
  ii. Cracking Codes with Python (2018)
  iii. Beyond the Basic Stuff with Python (2020)
- All of them are also freely available online.
- freeCodeCamp.org has plenty of wonderful YouTube-based tutorials: you can use Google Colab to go through them and create your own notebooks as you learn!
- Lastly, I have created a Google Chat, `PythonGame`, which I will use for my "Game programming with Python" course this fall: let me know if you wish to be invited. I use this channel to share.

Quizzes

Quizziz Library:

1. Python Basics Quiz (10 questions) - after the first session
2. Python Programming Quiz (8 questions) - after the second session

Footnotes

[fn:3] Packages like `matplotlib` or `random` are also called *libraries*, and sub-packages like `pyplot` in `matplotlib` are called *modules*. Either of these three terms (package, library, module) will do.

[fn:2]This goes back to the first proper programming manual written by Kernighan and Ritchie for the C programming language, and it stuck (this seminal book also bot its own Wikipedia page).

[fn:1]The use of `spam`, `ham`, `bacon` and `eggs` is an hommage to the origin of the name for the Python language, the British comedy group 'Monty Python' - see also their "Spam" sketch, which is so famous that it has got its own Wikipedia page!