Python Tutorial

Basic course - 19. und 20. November 2024

SFB 1328

David Lohr d.lohr@uke.de
Fatemeh Hadaeghi f.hadaeghi@uke.de
René Werner r.werner@uke.de

Agenda

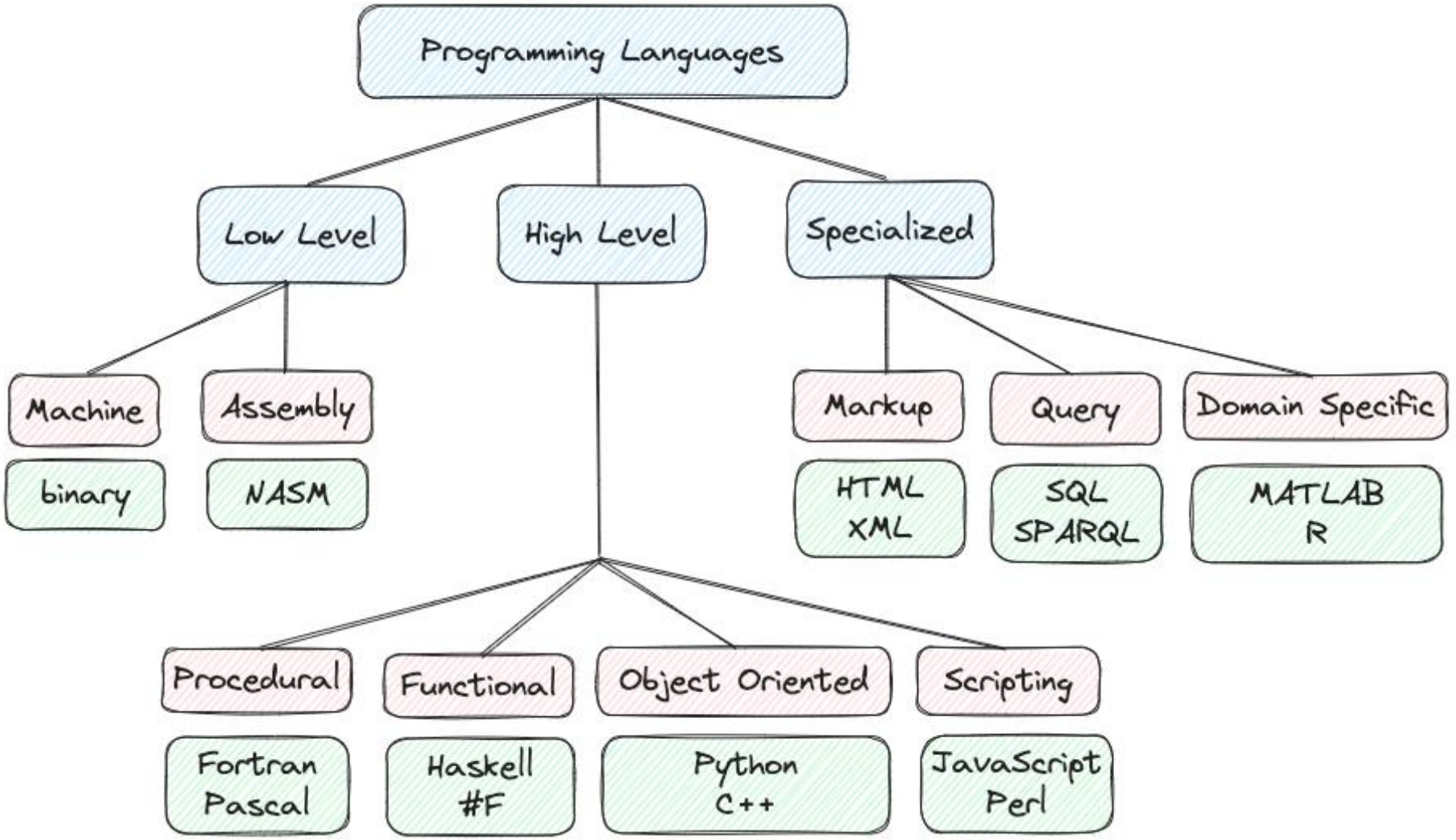
Tuesday, November 19th, 2024

09:00 – 12:00 Python basics, hands-on
12:00 – 13:00 Break
13:00 – 15:00 Python basics , hands-on, Chat GPT

Wednesday, November 20th, 2024

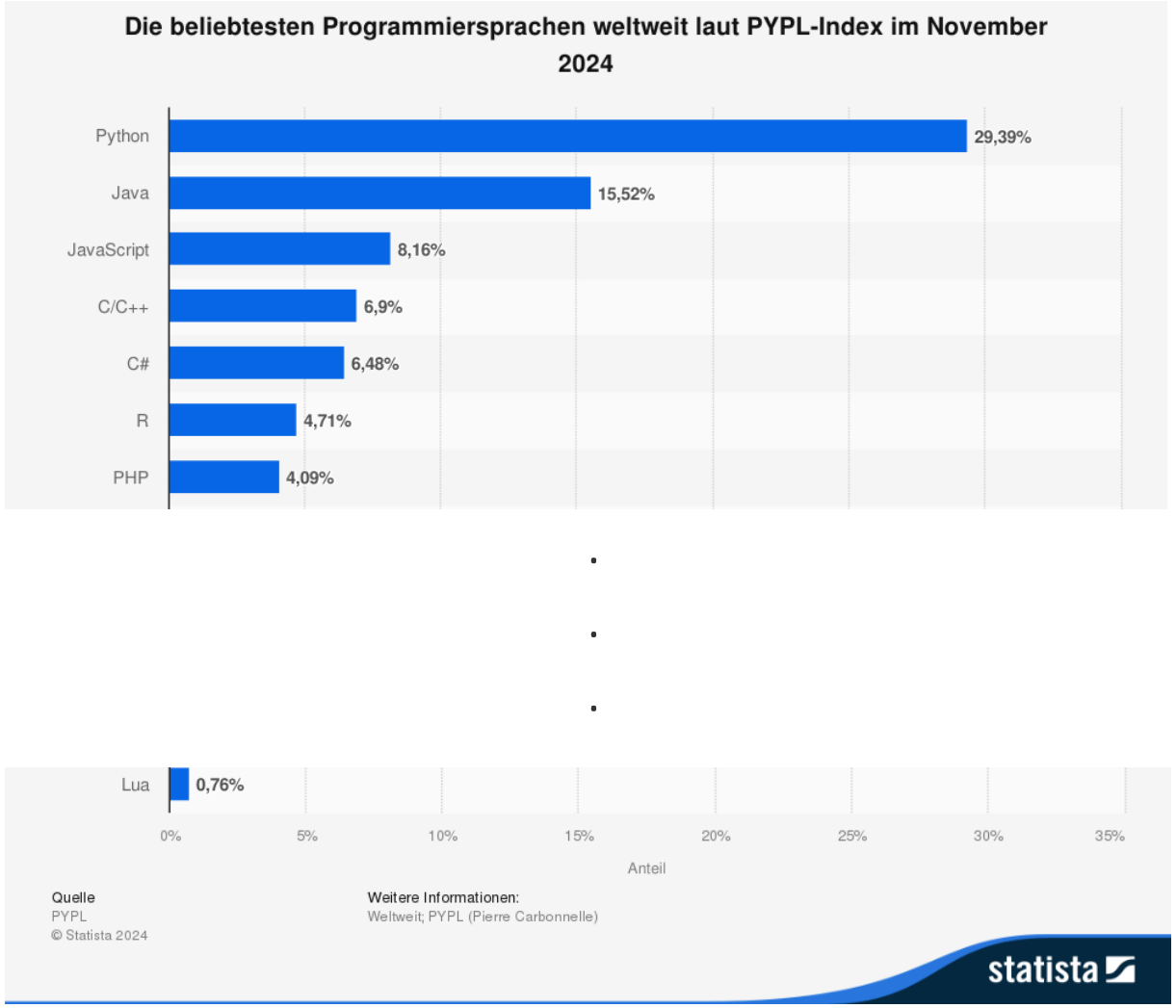
09:00 – 12:00 Image processing basics
12:00 – 13:00 Break
13:00 – 15:00 Hands-on

Why Python?


Programming Languages
Low Level
High Level
Specialized
Machine
Assembly
Markup
Query
Domain Specific
binary
NASM
HTML
XML
SQL
SPARQL
MATLAB
R
Procedural
Functional
Object Oriented
Scripting
Fortran
Pascal
Haskell
#F
Python
C++
JavaScript
Perl

Why Python?

- License free
- Independent of operating system
- Easy to read and write
- Universal, many different applications
- Large community, many modules
- Interpreted language



Content overview

- Data types and operators
- Data structures
- Control Flow
- Functions
- Classes
- Scripting

Data types and operators

Primary data types

Type	Example
Integer (int)	1, 2, 3, 100
Float	1.5, 2.1, 3.641, 0.5
Character (char)	‘a’, ‘b’, ‘c’, ’z’
String (str)	‘ hello world’
Logical (Boolean)	True, False

Data types and operators

Arithmetic operators

Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulo (Mod)	%
Exponentiation	**
Divide and round down to nearest integer	//

Input	Output:
print(3+5-2)	6
print(3*3/2)	4.5
print(3**3)	9
print(9%2)	1

Data types and operators

Variables and assignment operators

=	Assignment
+=	Assignment +1
-=	Assignment -1

Input	Output:
x, y, z = 4, 5, 6	
print(x)	4
print(y)	5
print(z)	6
x += 1	5
y += y	10

Data types and operators

Booleans, comparison operators, and logical operators

- The **bool** holds one of the values True or False
- often encoded as 1 or 0, respectively

Operator	Operation	Example	Output
<	Less than	3<5	True
>	Greater than	3>5	False
<=	Less than or equal	3<=3	True
>=	Greater than or equal	3>=3	True
==	Equal to	3==5	False
!=	Not equal to	3!=5	True

Operator	Operation	Example	Output
And	Evaluates if all statements are true	5>3 and 5==5	True
Or	Evaluates if at least one statement is true	5<3 or 12>1	True
Not	Inverts the bool value	Not 5<3	True

Data types and operators

Strings

- can be defined using double and single quotes “ ”, and ‘ ’

```
>>> a_string = “this is a string”
>>> another_string=‘this is also a string’
```

- Quotations within a string require ‘\’

```
>>> a_string = ‘Peter\’s cell count was way off’
>>>print(a_string)
Peter’s cell count was way off
```

- can be indexed into like an array

```
>>>first_word[0]
c
>>>first_word[-1]
l
```

```
>>> first_word = ‘cell’
>>> second_word = ‘count’
>>> print(first_word + second_word)
cellcount

>>> print(first_word + ‘ ‘ + second_word)
cell count

>>> print(first_word * 5)
cellcellcellcellcell

>>>first_word!=second_word
True
```

Data structures

Strings

- Functions can be applied to strings

```
>>> a_string = ‘this is a string’
>>> len(a_string)
16
>>> print(a_string)
this is a string
```

- Strings can be formatted using variables

```
>>> print(“{} cells have been counted”.format(27)
27 cells have been counted

>>>a=27
>>> print(f”{a} cells have been counted”)
27 cells have been counted
```

- Strings have so-called methods attached to them

```
>>> a_string.islower()
True
>>> a_string.count(‘s’)
3
>>> a_string.find(‘s’)
2
```

Data structures

Lists

- A list is a container that can store elements of different kinds

```
>>> short = [‘counting’, 10, ‘cells’]
>>> type(short)
list
>>> mixed = [‘more’,1,4.12,short]
[‘more’,1,4.12,[‘counting’, 10, ‘cells’]]
```

- Lists can be indexed like strings

```
>>> short[0]
‘counting’
>>>short[-1]
‘cells’
```

- Lists can be adjusted, they are mutable

```
>>> short[1] = ‘cells’
>>>short[2] = ‘is’
>>>short[3] = ‘not’
>>>short[4] = ‘fun’
```

- Lists have methods attached to them

```
>>> result = [1,5,10]
>>>result.append(15)
>>>result[1,5,10,15]
>>>result.remove?                # ? For documentation
```

- Tuples are like lists, but nor mutable!

Data structures

Sets

- Container that will have each element only once

```
>>> unique = {1,2,2,2,2,3}
>>> unique
{1,2,3}
```

- Strings have methods which they can use

```
>>> unique.add(4)
>>> unique
{1,2,3,4}
```

- Sets do not preserve insertion order

```
>>> ordered = {‘counting’, ’cells’, 5,1}
>>> ordered
{1,5,’cells’,’counting}
```

- Sets are faster regarding compute time for lookups

Data structures

Dictionaries

- A dictionary is a mapping from keys to values

```
>>> word2num = {
                “couting”: 1,
                “cells”: 2,
                “is”: 3,
                “not”: 4,
                “fun”: 5
                    }
>>> word2num
“couting”: 1, “cells”: 2, “is” : 3, “not” : 4, “fun”: 5}
```

- Dictionary can be accessed like this:

```
>>> word2num[“cells”]
2
```

- Dictionaries can be nested

```
>>> nested = {
            “first”:{
                “one”: 1,
                “two”: 2,
                },
            “second”: {
                “three”: 3,
                “four”: 4,
                },
            }
>>> nested[“second”][“four”]
4
```

- Keys are unique, values can be anything

Data structures

Dictionaries

- Dictionaries can be built from key-value pairs

```
>>> items = [   (“one”: 1),
                (“two”: 2),
                (“three”: 3),
                ]
>>> pairs = dict(items)
>>> pairs, type(pairs)
({“one”: 1, “two”: 2, “three”: 3}, dict)
```

- We often want to iterate through dictionaries

```
>>> for key in pairs:
            print(key)
one
two
three

>>> for key in pairs.keys():
            print(key)
one
two
three

>>> for vals in pairs.values():
            print(vals)
1
2
3
```

Control flow

If statements

- Is a conditional statement that runs or skips code, if the condition is met

```
>>> if cell_number < 5:
        print('Do another measurement')
        cell_number += 10
        hours_spent_measuring += 2
```

- Multiple conditions may be combined

```
>>> if cell_number < 5 < 50
        print('Do another measurement')
        cell_number += 10
        hours_spent_measuring += 2
```

```
>>> if season == 'spring':
            print('plant the garden!'
    elif season == 'summer'
            print('water the garden!'
    elif season == 'autumn':
            print('harvest ripe fruits!')
    elif season == 'winter':
            print('stay indoors!')
    else:
            print('unknown season')
```

Control flow

For loops

- Python has two kinds of loops, the for loop is used more frequently. It is a definite iteration, meaning it runs a pre-defined number of times

```
>>> cells = [‘stem cell’, ‘neuron’, ‘skin cells’, ‘blood cells’]
>>> for cell in cells:
        print(cell)
stem cell
neuron
skin cell
blood cell
```

- range() is a built-in python function used to create iterables:
  range(start=0, stop, step1)

```
>>> for i in range(3):
        print(“cell”)
cell
cell
cell
```

```
>>> for number in range(2, 7, 2):
        print(number)
2
4
6

>>> numbers = [0, 5, 10]
>>> for i, number in enumerate (numbers)
        print(i, number)
0 0
1 5
2 10

>>> dict(enumerate(numbers))
{0: 0, 1: 5, 2: 10}
```

Control flow

While loops

- While loops run as long as the condition is true. It is an Indefinite iteration, meaning it repeats an unknown number of times

```
>>> cell_counts = [0, 5, 10, 7, 14, 28]
>>> collection = []
>>> while sum(collection) < 25:
        collection.append(cell_counts.pop())
>>> collections
[0, 5, 10, 7, 14]
```

```
>>> cell_counts = [0, 5, 10, 7]
>>> all_counts  = []
>>> while cell_counts:
        cell_number = cell_counts.pop()
        all_counts.append(cell_number)
        print(f"adding {cell_number}, total count:
              {sum(all_counts)}")

adding 0 cells, total count: 0
adding 5 cells, total count: 5
adding 10 cells, total count: 15
adding 7 cells, total count: 22
```

Control flow

Break, continue

- Break terminates the loop

```
>>> for cell_count, cell_type in protocol:
        print(f“current cell count: {cell_count}”)
        if cells >= 100:
            print(“breaking from loop, count high enough”)
            break
        elif cells + cell_count > 100:
            print(f“skipping {cell_type} ({cell_count} cells”)
            continue
        else:
            print(f” adding {cell_type} ({cell_count} cells”)
            items.append(cell_type)
            cells+= cell_count
```

Control flow

List comprehension

- List comprehensions are quick and concise methods to generate lists

```
>>> result = []
>>> for i in range(10):
        if i>5:
            result.append(i)
>>> result
[6,7,8,9]
```

- The above statement can be reduced to:

```
>>> result = [i for i in range(10) if n>5]
```

- Enables various kinds of combinations

```
>>> result = [i if i>5 else “smaller” for I in range (10)]
>>> result
[“smaller”, “smaller”, “smaller”, “smaller”, “smaller”, “smaller”, 6, 7, 8, ,9]
```

- Comprehensions also work for:

Tuples, sets, dictionaries

Functions

Defining functions

- A function definition requires a name and potentially any input variables as well as the code it is supposed to apply

```
>>> def DNA2RNA(DNA):
        """
        converts DNA string to RNA string.
        Parameters: DNA sequence as string
        Return: RNA sequence as a string
        """
        transliterate = DNA.maketrans('tT','uU')
        RNA = DNA.translate(transliterate)
        return RNA
```

- A defined function can then be applied. This reduces the amount of code for often repeated operations

```
>>> zika_DNA  = 'AGTTGTTGATCTGTGTGAGTCAGACTGCG'
>>> zika_RNA = DNA2RNA(zika_DNA)
>>> print(zika_RNA)
'AGUUGUUGAUCUGUGUGAGUCAGACUGCG'
```

Functions

Variable scope

- Refers to which parts of a program a variable can be referenced or used from

```
>>> def some_function():
        word = “cell’

>>> print(word)                    # will create error
```

```
>>> def some_function():           # these two functions work fine
        word = ‘cell’
>>> another_function():
        word = ‘count’
```

```
>>> word = ‘cell’
>>> def some_function():
        print(word)                 # will work fine
```

Functions

Lambda expressions

- Lambda expressions create anonymous functions which are functions without a name
- They are useful when creating short functions that are not used later in the code
- Particularly useful for higher order functions or functions that take in other functions as arguments

```
>>> def multiply(x,y)
        return x*y

>>> multiply = lambda x, y: x*y
```

- Both functions are used in the same way and can be called like this:

```
>>> multiply(4,7)
```

Scripts

Importing local or self-written scripts

- Lambda expressions create anonymous functions which are functions without a name
- They are useful when creating short functions that are not used later in the code
- Particularly useful for higher order functions or functions that take in other functions as arguments

```
>>> def multiply(x,y)
        return x*y

>>> multiply = lambda x, y: x*y
```

- Both functions are used in the same way and can be called like this:

```
>>> multiply(4,7)
```

Scripts

Importing from third party libraries

- Helpful when working on bigger projects keep your code clean and concise by organizing code into multiple files for reuse

```
## demo.py

>>> import useful_functions as uf

>>> cell_counts = [25, 85, 83, 56, 91]

>>> mean = uf.mean(cell_counts)
>>> curved = uf.add_five(cell_counts)
>>> mean_c = uf.mean(curved)

>>> print("Counts: ", cell_counts)
[25, 85, 83, 56, 91]

>>> print("Original mean: ", mean, " new mean: ", mean_c)
Original mean: 68, new mean: 73
```

```
## useful_functions.py

>>> def mean(num_list):
        return sum(num_list) / len(num_list)

>>> def add_five(num_list)
        return [n+5 for n in num_list]
```

Importing from third-party libraries

- A lot of useful functions are already available in third party libraries, e.g. numpy, scipy, pandas, etc.

- They can be installed via “pip install”

```
>>> pip install package_name
```

- Better is to install such libraries using a library manager like conda, or mamba

```
>>> conda install package_name
>>> mamba install package_name        # alternative to conda
```

- For larger projects with various different third-party libraries requirements.txt files are available
- These files list all the libraries used within a project, which allows direct installation of all required libraries

```
>>> pip install –r requirements.txt
```

```
## requirements.txt
Numpy==2.1.3
Fastai==2.1.1
Pytorch==2.5.0
```

Happy coding ☺

Exercise 1

Create a program that asks the user to enter their name and their age. Print out a message addressed to them that tells them the year that they will turn 100 years old

Exercise 2

Implement a function that takes as input three variables and returns the largest of the three.

Exercise 3

Take a list, say for example this one:

```
List = [1, 8, 21, 34, 5, 2, 55, 89, 3]
```

and write a program that prints out all the elements of the list that are less than 10.

Extras:

1) Instead of printing elements one by one: make a new list that has all the elements less than 5 in it and print it
2) Write this in one line
3) Ask the user for a number and return a list that contains only elements from the original list that are smaller than the given number

Exercise 4

Write a program that asks the user how many Fibonnaci numbers to generate and then generates them

Exercise 5

Write a rock paper scissors game for two players. The choice of rock, paper, or scissor will be given via keyboard input.

Exercise 1: solution

```
>>> Name = input(“What is your name?“)
>>> Age = input(“How old are you?“)
>>> year = 2024 - Age + 100
>>> print(f“ You will be 100 years old in the year: {year}”)



# alternative
>>> Import datetime

>>> Name = input(“What is your name?“)
>>> Age = input(“How old are you?“)
>>> year = datetime.now().year - Age + 100
>>> print(f“ You will be 100 years old in the year: {year}”)
```

Exercise 2: solution

```
>>> def max_of_three(var_1,var_2,var_3):
        highest = 0
        if  var_1>var_2 and var_1>var_3:
            highest = var_1
        elif var_2 > var_1 and var_2>var_3:
            highest = var_2
        else:
            highest = var_3
        return(f”The highest value is: {highest}”)
```

Exercise 3: solution

```
>>> List = [1, 8, 21, 34, 5, 2, 55, 89, 3]

# basic problem:
>>> for x in List:
        if x<10: print(x)

# combine with extras 1 and 2
>>> print([x for x in List if x<10])

# extra 3
>>> number = input("Put in a number for which you want to list all smaller numbers")
>>> print([x for x in List if x<number].sort())
```

Exercise 4: solution

```
>>> def generate_fibo():
        count = int(input("How many numbers should I generate of the Fibonacci series"?))
        k = 1
        if count == 0:
            fib = []
        elif count == 1:
            fib = [1]
        elif count == 2:
            fib = [1,1]
        else count > 2:
            fib = [1,1]
            while k < (count -1):
                fib.append(fib[k] + fib[k-1])
                k +=1
        return fib
```

Exercise 5: solution

```
>>> Player_1 = input(“Player 1, do you choose rock, paper, or scissors?“)
>>> Player_2 = input(“Player 2, do you choose rock, paper, or scissors?“)

>>> def rock_paper_scissors(u1,u2):
        if u1 == u2:
            return(“t is a tie, play again!)
        elif u1 == ‘rock‘:
            if u2 == ‘scissors’:
                return(“Player 1 wins with rock”)
            else:
                return(“Player 2 wins with paper”)
        elif u1 == ‘scissors‘:
            if u2 == ‘paper’:
                return(“Player 1 wins with scissors”)
            else:
                return(“Player 2 wins with rock”)
        elif u1 == ‘paper‘:
            if u2 == ‘rock’:
                return(“Player 1 wins with paper”)
            else:
                return(“Player 2 wins with scissors”)
        else:
            return(“Invalid input, I am missing at least one rock, paper, or scissor”)

>>> print(rock_paper_scissors(Player_1,Player_2))
```