CEPLAS
Cluster of Excellence on Plant Sciences

Introduction to Python week 3

QBio104

Material modified from M. Röttger and A. Schrader

Recap

String formatting

```
#Here we will show how to use fstrings
#F strings are declared (similarly to raw strings) using an F or f as the first letter in a print statement.
#We then place any variables we want in the print statement within curly parentheses {}
species = 'Homo Sapiens'
name = 'Vittorio'

#the two print statements will print the same message to standard output
print ("My name is " + name + " and I am an " + species)
print (f"My name is {name} and I am an {species}")
```

```
My name is Vittorio and I am an Homo Sapiens
My name is Vittorio and I am an Homo Sapiens
```

```
#Here we will have another example of fstrings
#There are multiple ways to use format strings, choose the one that is more intuitive for you and stick to it.

species_a_count = 10
species_b_count = 6
species_c_count = 3

print (f"There are {species_a_count + species_b_count + species_c_count} individuals in the environment")
print ('There are {} individuals in the environment'.format(species_a_count + species_b_count + species_c_count))
print ('There are {total} individuals in the environment'.format(total=species_a_count + species_b_count + species_c_count))
```

```
There are 19 individuals in the environment
There are 19 individuals in the environment
There are 19 individuals in the environment
```

Recap

Reversing a string, find, replace

- The slicing operator `[start:stop]` can be used to access certain parts of a character string.
- To reverse a string with the index we use `str[::-1]`
- `str.find('substring')` returns the index of the first occurrence of the substring in the string
- `str.replace('substring1', 'substring2')` replaces all occurrences of `substring1` with `substring2`

Recap

Conditional statements

```
if CONDITION:
    BLOCK
elif CONDITION:
    BLOCK
else:
    BLOCK
```

```python
#Let's create a function that orders at a restaurant based on diet and budget

def restaurant_order(budget, diet):
    """This function suggests an order based on diet and budget

    var budget (int or float): Amout of money, in euro
    var diet (str): It gives 'Vegetarian' option when selecting 'Vegetarian'

    returns (str): suggested order
    """
    if budget > 50:
        if diet == 'Vegetarian':
            order = 'Luxurious charcuterie board with vegetables and cheese, dessert, red wine glass, cappuccino'
        else:
            order = 'Tuna steak, red wine glass, dessert'
    elif budget >= 20:
        if diet == 'Vegetarian':
            order = 'Aloo gobi curry with rice, aloo pakora, red lentil dahl and mango lassi'
        else:
            order = 'Lamb curry with naan, malai kofta, masala chai'
    elif budget > 5:
        if diet == 'Vegetarian':
            order = 'Pommes with mayo, fritz-limo'
        else:
            order = 'Currywurst, fritz-kola'
    else:
        order = 'Tap water'
    return order
```

```python
diet = input("What is your diet?")
budget = float(input("What is your budget?"))

order = restaurant_order(budget, diet)
print (f'You could order {order}')
```

```
What is your diet? Vegetarian
What is your budget? 40
You could order Aloo gobi curry with rice, aloo pakora, red lentil dahl and mango lassi
```

Introduction to python

Topics

- Logical operators for comparison
- Logical connectives
- Propositional calculus and laws
- Control structures: Loops
- Flow diagrams
- Application of loop constructs
  - Fibonacci sequence

Logical operators

logical operators for comparison

Operators, that return Boolean values, are called logical operators.

We already know the following operators for comparison:

```
#Here we will test comparison operators

maize_root_length = 160
wheat_root_length = 100
rice_root_length = 60

print (maize_root_length == wheat_root_length)
print (maize_root_length != rice_root_length)
print (wheat_root_length > maize_root_length)
print (rice_root_length + wheat_root_length <= maize_root_length)
```

```
False
True
False
True
```

Operator	Operation
==	Equal
!=	Not equal
<	Less than
>	Greater than
<=	Less or equal to
>=	Greater or equal to

Logical connectives

`and`, `or`, `not`, `xor`

- Logical connectives `and`, `or`, `not` and `xor` (E`x`CLUSIVE `or`) can be used to logically connect Boolean input values with a Boolean output value.
- Using logical connectives, very complex logical expressions can be build up.

`and` needs both terms being compared to be `True`

`or` need either or both the terms being compared to be `True`

`not` refers to one term and is `True` only when the term is `False` (0 or an empty `str` is considered `False` in Boolean operators)

`xor` need either (not both!) the terms being compared to be `True`

Logical connector truth table

Given A and B, what is the result of the statement

A	B
False	False
False	True
True	False
True	True

A == B
True
False
False
True

A != B
False
True
True
False

A and B
False
False
False
True

A or B *
False
True
True
True

not A
True
True
False
False

```
#Show the results of truth table

T = True
F = False

print (T and T)
print (T and F)
print (T or F)
print (F or F)
print (not F)
```

```
True
False
True
False
True
```

```
#Show the results of truth table using non-booleans
#Using conditional operators

T = "False"
F = 0

if T and F:
    print (True)
else:
    print (False)

if T or F:
    print (True)
else:
    print (False)

if not F:
    print (True)
else:
    print (False)
```

```
False
True
True
```

Logical operators

Examples with `and`

```
#Here we show the and operator

import random

sunny = False
warm = False

if random.random() > 0.5:
    sunny = True

if random.random() > 0.5:
    warm = True

if sunny and warm:
    print ("It's sunny and warm!", end = '')
else:
    print ("Just a regular day in Germany", end = '')

print (" What were the chances?")
```

```
It's sunny and warm! What were the chances?
```

A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

Logical operators

Examples with `or`

```
#Here we show the or operator

import random

sunny = False
warm = False

if random.random() > 0.5:
    sunny = True

if random.random() > 0.5:
    warm = True

if sunny or warm:
    if sunny and warm:
        print ("it's sunny and/or warm! ", end = '')
    elif sunny:
        print ("it's sunny but cold! ", end = '')
    elif warm:
        print ("It's cloudy but warm! ", end = '')
else:
    print ("Who cares, are you made of sugar? ", end = '')

print (" What were the chances?")
```

```
It's cloudy but warm!  What were the chances?
```

A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

Logical operators

Examples with `not`

```
#here we show the not operator

coin_is_heads = False

if random.random() > 0.5:
    coin_is_heads = True

print (coin_is_heads)
print (not coin_is_heads)
print (not not coin_is_heads)

if not coin_is_heads == True:
    print ('Tails!')
else:
    print ('Heads!')

if coin_is_heads != True:
    print ('Tails!')
else:
    print ('Heads!')
```

```
True
False
True
Heads!
Heads!
```

A	Not A
True	False
False	True

Combining multiple conditional operators

Operators are solved based on a hyerarchy system

1. Arithmetic operators (+, -, *, etc..)
2. Comparison operators (==, >, >=, etc..)
3. Logical operators (and, or, not)
4. Assignment operators

Still, you should ALWAYS use parentheses when using multiple operators for clarity.

```
#How to use multiple operators

T = True
F = False
T1 = 1
F1 = 0

if T == T1 or F and F1:
    print ("It's hard to interpret!")

if (T == T1) or (F and F1):
    print ("The parentheses don't always alter the result")

if T == (T1 or F) and F1:
    print ("But sometimes they do")
```

```
It's hard to interpret!
The parentheses don't always alter the result
```

Hot question

More like a riddle

```
#Hot question on boolean logic and operators

A = "0"
B = bool(0)

if not A:
    B = "1"
    if A and B:
        print ("Well.. that was easy!")
    elif type(B) == type('string'):
        print ("You almost got me there!")
    else:
        print ("Nah, you can't fool me")
elif not B and A:
    if bool(int(A)) or B:
        print ("I think I got it")
        if bool(int(A)) and B:
            print ("oh, nevermind")
        elif bool(int(A)):
            print ("Yes, I do!")
        elif B:
            print ("No, I don't..")
    if not not not (bool(int(A))):
        print ("I have no idea, I can't solve this")
else:
    print ("What is a boolean again?")
```

CEPLAS
Cluster of Excellence on Plant Sciences

Introduction to Python

Break – 15 minutes

Hot question

Answer

```
#Hot question on boolean logic and operators

A = "0"
B = bool(0)

if not A:
    B = "1"
    if A and B:
        print ("Well.. that was easy!")
    elif type(B) == type('string'):
        print ("You almost got me there!")
    else:
        print ("Nah, you can't fool me")
elif not B and A:
    if bool(int(A)) or B:
        print ("I think I got it")
        if bool(int(A)) and B:
            print ("oh, nevermind")
        elif bool(int(A)):
            print ("Yes, I do!")
        elif B:
            print ("No, I don't..")
    if not not not (bool(int(A))):
        print ("I have no idea, I can't solve this")
else:
    print ("What is a boolean again?")
```

```
I have no idea, I can't solve this
```

Logical laws

Laws for logical connectives

- Associative laws:

$$(a \texttt{ and } b) \texttt{ and } c = a \texttt{ and } (b \texttt{ and } c)$$

$$(a \texttt{ or } b) \texttt{ or } c = a \texttt{ or } (b \texttt{ or } c)$$

- Commutative laws:

$$a \texttt{ and } b = b \texttt{ and } a$$

$$a \texttt{ or } b = b \texttt{ or } a$$

- Distributive laws:

$$a \texttt{ or } (b \texttt{ and } c) = (a \texttt{ or } b) \texttt{ and } (a \texttt{ or } c)$$

$$a \texttt{ and } (b \texttt{ or } c) = (a \texttt{ and } b) \texttt{ or } (a \texttt{ and } c)$$

Logical laws

Laws for logical connectives

- Absorption laws:

$$a \texttt{ or } (a \texttt{ and } b) = a$$

$$a \texttt{ and } (a \texttt{ or } b) = a$$

- De Morgan laws:

$$\texttt{not } (a \texttt{ and } b) = (\texttt{not } a) \texttt{ or } (\texttt{not } b)$$

$$\texttt{not } (a \texttt{ or } b) = (\texttt{not } a) \texttt{ and } (\texttt{not } b)$$

Loops

Another form of control structures

- A loop is used, if a block of statements is supposed to be executed repeatedly.

- In python you can find `for` and `while` loops

- `for` loops are executed a set amount of times, `while` loops repeat as long as the condition is met. By design, `while` loops are prone to endless loops (not good!). Therefore, we should and will use `for` loops when possible.

`for`-loop

Deterministic loop

- A `for`-loop is best used, if the number of desirable loop iterations is known in advance.

- Syntax of the `for`-loop :

```
for LOOPVARIABLE in SEQUENCE:
    BLOCK
```

- Sequence can be an arbitrary iterable object (e.g.: `str`, `list`, `tuple`).

`for`-loop

How it works

- The loop iterates over the single elements in a sequence.
- In each iteration, the loop variable references the next object in the sequence type object.
- The function `range(…)` can be used to generate a sequence of integer numbers that can be used as sequence object in a loop.
- Beyond the examples given here, we will always use the `range(…)` function. As it provides a consistent sequence that iterates over a variable with a known type (integer)

The `range(...)` function

Get used to it!

- The function `range(`**start, stop, step**`)` can be used to generate a sequence of integer numbers.
- **start** gives the starting number. If omitted, 0 is used.
- **stop** gives the end of the sequence. stop is not contained in the sequence.
- **step** gives the difference between elements in the generated sequence.
- It returns a “range” object which can be iterated over in a `for`-loop

What numbers will be in `range(1,10,3)`?

for-loop examples

Iterating over sequences

```
#here we show how to use for-loops

my_DNA = 'ATGCATGC'
for nucleotide in my_DNA:
    print (nucleotide)
```

```
A
T
G
C
A
T
G
C
```

```
#here we show how to use for-loops, using the range function

my_DNA = 'ATGCATGC'
for nucleotide_position in range(len(my_DNA)):
    print (nucleotide_position, my_DNA[nucleotide_position])
```

```
0 A
1 T
2 G
3 C
4 A
5 T
6 G
7 C
```

```
#here we show how to use for-loops, using the range function and the step

for nucleotide_position in range(1, len(my_DNA), 2):
    print (nucleotide_position, my_DNA[nucleotide_position])
```

```
1 T
3 C
5 T
7 C
```

`while`-loop

Condition-based loop

- Before each iteration of a `while`-loop, a condition is evaluated
- If the respective condition is `True`, the while-block will be executed.
- Once executed, the condition is evaluated again.. And so on.
- If the initial condition is `False` the loop will not be executed at all

Syntax of the `while`-loop:

```
while CONDITION:
    BLOCK
```

while-loop examples

One of the example is an “infinite loops”, can you spot it?

```
#Here is a while loop, is it infinite?

my_DNA = "ATGCATGC"
nucleotide_position = 0

while nucleotide_position < len(my_DNA):
    print (my_DNA[nucleotide_position])
    nucleotide_position += 2
```

```
#Here is a while loop, is it infinite?

while len(my_DNA)!= 0:
    my_DNA.replace(my_DNA[0], '')
    print (my_DNA)
```

```
#Here is a while loop, is it infinite?

species_count = 0
while species_count != 10:
    species_count += 2
    print (species_count)
    species_count -= 1
```

while-loop examples

One of the example is an “infinite loops”, can you spot it?

```
#Here is a while loop, is it infinite?

my_DNA = "ATGCATGC"
nucleotide_position = 0

while nucleotide_position < len(my_DNA):
    print (my_DNA[nucleotide_position])
    nucleotide_position += 2
```

```
A
G
A
G
```

```
#Here is a while loop, is it infinite?

while len(my_DNA)!= 0:
    my_DNA.replace(my_DNA[0], '')
    print (my_DNA)
```

```
ATGCATGC
ATGCATGC
ATGCATGC
ATGCATGC
ATGCATGC
ATGCATGC
ATGCATGC
ATGCATGC
ATGCATGC
ATGCATGC
ATGCATGC
ATGCATGC
ATGCATGC
ATGCATGC
ATGCATGC
ATGCATGC
ATGCATGC
ATGCATGC
ATGCATGC
```

```
#Here is a while loop, is it infinite?

species_count = 0
while species_count != 10:
    species_count += 2
    print (species_count)
    species_count -= 1
```

```
2
3
4
5
6
7
8
9
10
11
```

Nested loops

Loops within loops within loops..

- A loop BLOCK can also contain another loop. We call these nested loops.
- Notice that the inner loops are completely resolved first

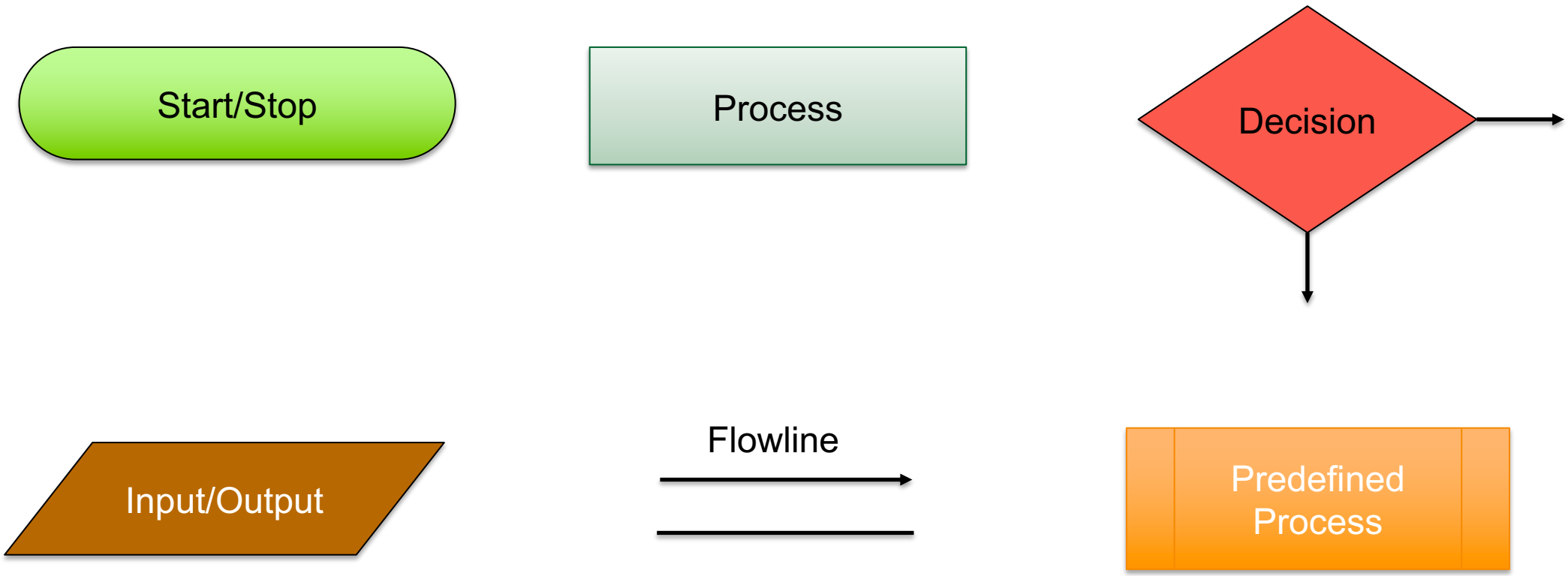
```
#Here we show how nested loops work

for first_multiplier in range(1, 7, 2):
    for second_multiplier in range(0, 6, 2):
        print (f"multiplying {first_multiplier} by {second_multiplier} equals {first_multiplier*second_multiplier}")
```

```
multiplying 1 by 0 equals 0
multiplying 1 by 2 equals 2
multiplying 1 by 4 equals 4
multiplying 3 by 0 equals 0
multiplying 3 by 2 equals 6
multiplying 3 by 4 equals 12
multiplying 5 by 0 equals 0
multiplying 5 by 2 equals 10
multiplying 5 by 4 equals 20
```

Flow diagrams

A visual representations of an algorithm



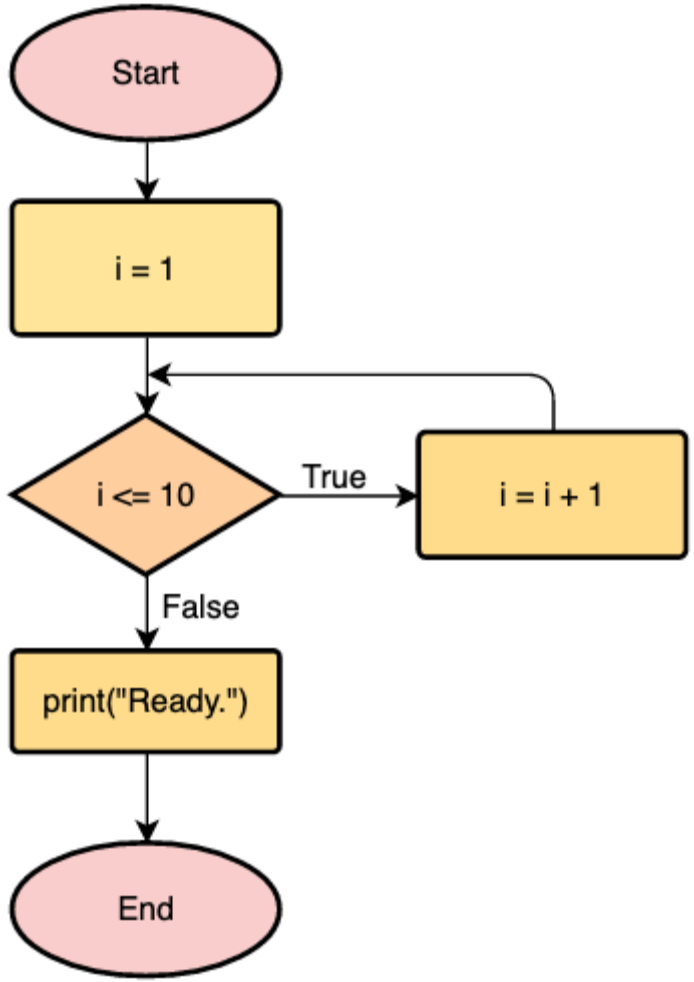
Flow diagrams

While loop example

```
#Here is the code for the flow diagram

i = 1
while i <=10:
    i+=1
print ("Ready.")
```

```
Ready.
```



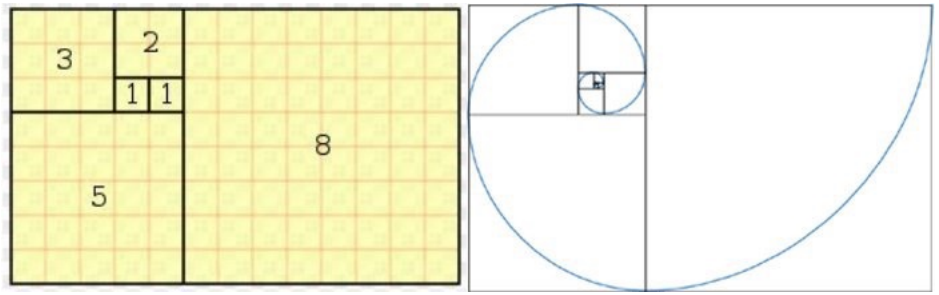
CEPLAS
Cluster of Excellence on Plant Sciences

Introduction to Python

Break – 5 minutes

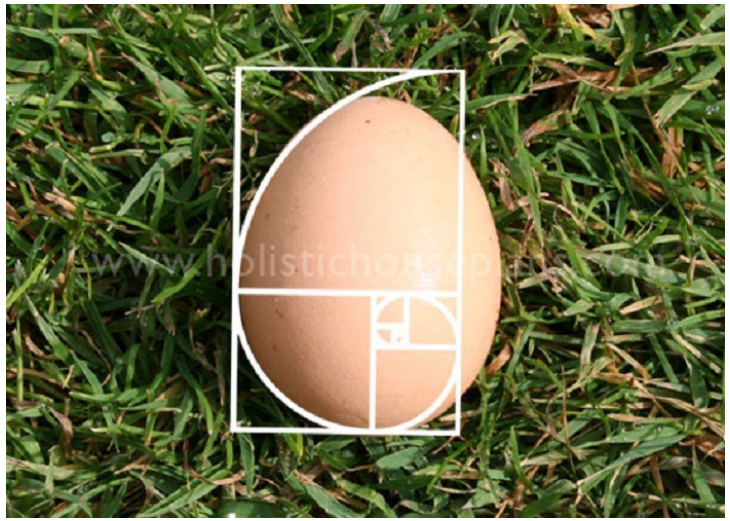
Application of loop constructs

Fibonacci sequence



Insteading.com/blog/fibonacci-sequence-in-nature

Jitze Couperus / flickr (Creative Commons)



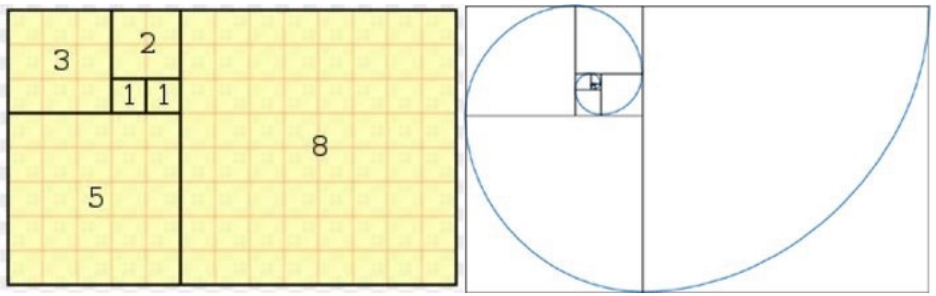
holistichouseplans.com

Aiko, Thomas & Juliette+Isaac Jitze Couperus / flickr (Creative Commons)

Fibonacci sequence

$F_n = F_{n-1} + F_{n-2}$

- A sequence of numbers (F_n)
  - $F_1 = 1$
  - $F_2 = 2$
  - $F_3 = 2$
  - $F_4 = 3$
  - $F_5 = 5$
  - $F_n = F_{n-1} + F_{n-2}$; for n>2



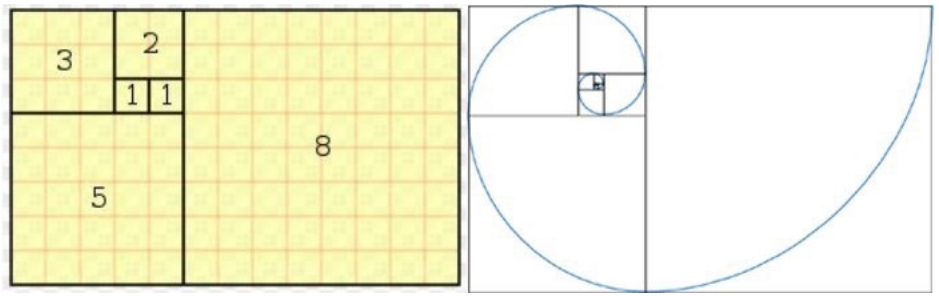
n	F_n
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
...	...

Fibonacci sequence

Adding F_{n-1} and F_{n-2} to the table

$F_n = F_{n-1} + F_{n-2}$; for n>2

n	F_n	F_{n-1}	F_{n-2}
1	1	-	-
2	1	1	-
3	2	1	1
4	3	2	1
5	5	3	2
6	8	5	3
7	13	8	5
8	21	13	8
...	...	...	...



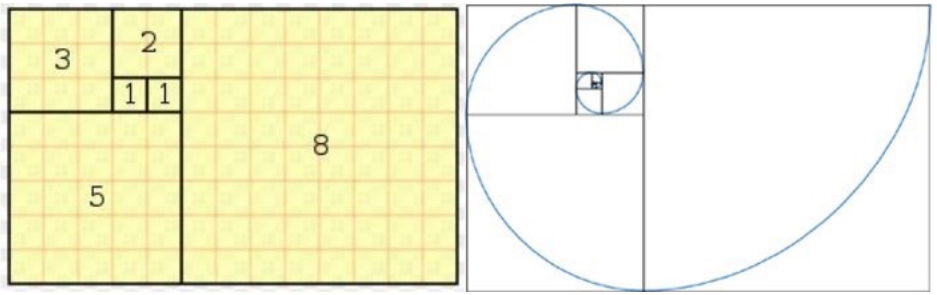
Fibonacci sequence

Understand the table and look at its structure

$F_n = F_{n-1} + F_{n-2}$; for n>2

- Fn-2 receives the value of F_{n-1} from the last row.
- Fn-1 receives the value of F_n from the last row.

n	F_n	F_{n-1}	F_{n-2}
1	1	-	-
2	1	1	-
3	2	1	1
4	3	2	1
5	5	3	2
6	8	5	3
7	13	8	5
8	21	13	8
...	...	...	...



Fibonacci sequence

Pseudo code to print the fibonacci sequence for a given N > 3

- What variables do we need to set?
- How do we create a loop?
  - What is the starting point?
  - What is the end?
- What operations have to happen in the loop?
- In which order?

n	F_n	F_{n-1}	F_{n-2}
1	1	-	-
2	1	1	-
3	2	1	1
4	3	2	1
5	5	3	2
6	8	5	3
7	13	8	5
8	21	13	8
...	...	...	...

Fibonacci sequence

Pseudo code to print the fibonacci sequence for a given N > 3

```
Starting with n = 3
Assign Fn to 2
Assign Fn_1 to 1
Assign Fn_2 to 1
for n1 in range(3, N):
    Assign Fn_2 to the last value of Fn_1
    Assign Fn_1 to the last value of Fn
    Assign Fn to Fn_1 + Fn_2

print (Fn)
```

n	F_n	F_{n-1}	F_{n-2}
1	1	-	-
2	1	1	-
3	2	1	1
4	3	2	1
5	5	3	2
6	8	5	3
7	13	8	5
8	21	13	8
...	...	...	...

Fibonacci sequence

Jupyter notebook implementation

```
"""
Fibonacci sequence, print the fibonacci N for N > 3 (eg. 8)
Here is the pseudocode:
Starting with n = 3
Assign Fn to 2
Assign Fn_1 to 1
Assign Fn_2 to 1
for n1 in range(3, N):
    Assign Fn_2 to the last value of Fn_1
    Assign Fn_1 to the last value of Fn
    Assign Fn to Fn_1 + Fn_2

print (Fn)
"""

def FibonacciN(N):
    """
    This function returns the Fibonacci number for a given N > 3
    """
    #Fn_2 is not necessary as it's set in the first iteration
    n = 3; Fn = 2; Fn_1 = 1; Fn_2 = 1

    for n1 in range(3, N):
        Fn_2 = Fn_1
        Fn_1 = Fn
        # this also works: Fn += Fn_2
        Fn = Fn_1 + Fn_2
    return Fn
```

[98]:

```
#Here we call the function we defined above

Fn = FibonacciN(8)
print (Fn)
```

```
21
```

[100]:

```
#We can also test it for much larger values

Fn = FibonacciN(1000)
print (Fn)
```

```
434665576869374564356885276750406258025646605173717804024817290895365554179
490518904038798400792551692959225930803226347752096896232398733224711616429
964409065331879382989696499285160037044761377951668492288 75
```

Summary

Logical connectors (and, or, not)

Loops (for, while)

Fibonacci sequence and algorithm