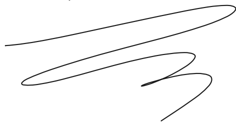
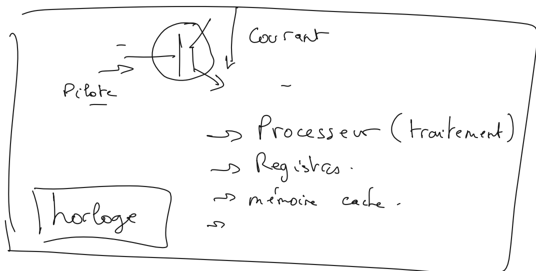
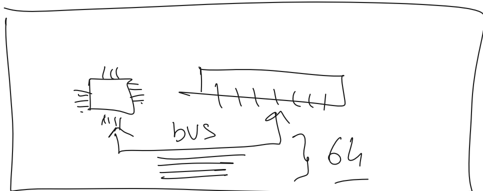
Bonjour tout le monde



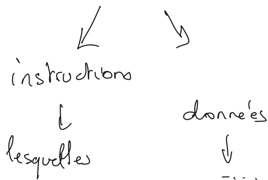
Origine de tout :

Silicium → Transistor





Processeur
(traiter)



Mémoire

binaire → 0/1

0/1 → 1 bit
8 bits → 1 octet
16 → mot
32 → mot long
64 → " "

64 bits
données

<octet<

0/1 → 2 états

00
01
10
11

} 4 état

000
001
010
011
100
101
110
111

} 8 états

2^n → n bits
= nb états

$2^8 = 256$

1 octet = 256 valeurs

instructions → valeurs numériques.

Code machine ↗

Instructions → nombres

Assembleur   Mnémoniques → Nombres -

C → 80 ⇒ COBOL
BASIC
Shell (Unix)

↓

C++ .

C + couche objet - mauvaises pratiques -

le langage le + puissant
+ performant → mémoire → vitesse
+ générateur de bugs -

SCRIPTING
(dev simple, éditable en ligne, peu performant)

↘

Php
groovy
Python

DEV. Applicatif
(dev. applications
→ phase compilation
→ + puissant, complexe)

→ C++
→ Delphi, C# (.net)
Java

Générations

1 Génération = Assembleur
2G = prog. naturelle (Spaghetti)
3G = Prog. structurée (pascal, C, php,)
4G = Prog. Orienté Objet (Delphi, C++, C#, Java)
5G = Prog. fonctionnelle (Scala, F#, ...)

Microsoft → - cmd (batch) .bat .cmd
(msdos) "équivalent" au bash

- VBscript → OObjet, complexe, pas fluide
- Powershell → puissant, carré, accessible

Linux: → shell (sh, bash, ksh, csh, ...)

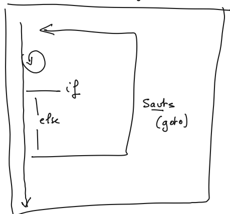
Universel: → (powershell)
perl, tcl, groovy (java)
Python

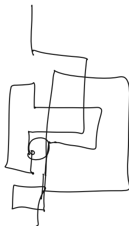
Scala

Programme:

instructions + variables

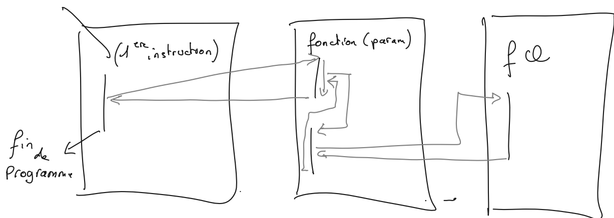
2G: fichier | 2G = spaghetti



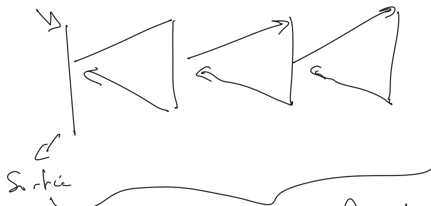


Minimum recommandé = Prog. structurée

Point d'entrée

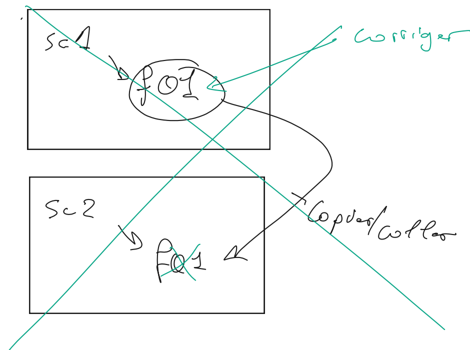


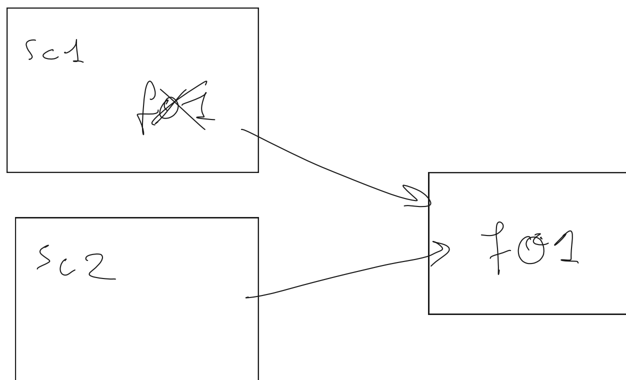
module = fichier de code



empilement de fonctions

Variable : stocke les données de certains types





```
#!/bin/bash

source bonjour.sh

# Atelier :
1) faites un script qui affiche un message via une fonction
2) modifiez pour déplacer votre fonction dans un autre fichier ( module )

say_hello2()
{
    echo "Hello World ! ";
}

# dire bonjour tout le monde
say_hello

say_hello2
say_hello2
say_hello2
say_hello2
```

---

contenu de bonjour.sh :

```
#!/bin/bash



# methode 1 : aliases
alias hello='echo "Hello
World ! "'

# methode 2 : fonctions
réutilisables

say_hello()
{
    echo "Hello World ! ";
}
```

1 module avec une fonction réutilisable en python :

cat a.py :

```
#!/usr/bin/python3

from b import *
test()
```

"depuis b.py, charge tout"

---

cat b.py

```
#!/usr/bin/python3

def test():
  print("ok")
```

```
ib@ib-VirtualBox:~/python$ ./a.py
ok
```

Passer un paramètre en python :

```
cat a.py
#!/usr/bin/python3

from b import *

test("Philippe")
```

Récupération de la valeur du paramètre :

```
cat b.py
#!/usr/bin/python3

def test( nom ):
  print("ok "+nom)
```

Avec argument :

```
root@ib-VirtualBox:/home/ib/python# ./a.py Laurent
ok Philippe
Avec l'argument :
ok Laurent

root@ib-VirtualBox:/home/ib/python# cat a.py
#!/usr/bin/python3

from b import *
import sys

test("Philippe")
print("Avec l'argument : ")
test(sys.argv[1])
```

Atelier :

Réalisez un script en python qui fasse l'addition des deux paramètres transmis en ligne de commande :
ex :
./add.py 10 5
15

Variante :
déclarez un fichier operateurs.py avec les fonctions
ajouter
soustraire
multiplier
diviser

qui fasse chaque opération (uniquement l'addition sera utilisée pour l'instant)
et seront utilisées par add.py

Hint : convertissez vos valeurs en numériques :

```
>>> a = "10"
>>> a
'10'
>>> ai = 10
>>> ai
10
>>> b = 5
>>> a+b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
>>> int(a)+int(b)
15
```

```
#!/usr/bin/python3
import sys     # importer le module système ( plein de fonctions sys.<quelque chose> )

def addition( ip1, ip2 ):
  return ip1+ip2       # renvoie à l'endroit de l'appel

def soustraction( ip1, ip2 ):
  return ip1-ip2

def division( ip1, ip2 ):
  return ip1/ip2

def multiplication ( ip1, ip2 ):
  return ip1*ip2

# et en faire une addition
n1 = int(sys.argv[1])   # le second ( commence a 0 ) argument en ligne de commande
n2 = int(sys.argv[2])   # mettre int( )   renvoie sous forme d'entier le paramètre qui lui est
passé

resultat = addition( n1, n2 )
print(" Le résultat est : "+str(resultat) )
```

Exemple simple :

```
#!/usr/bin/python3

import sys

def multiplier_par_coef( valeur, coefficient ):
  return valeur*coefficient

monresultat = multiplier_par_coef( int(sys.argv[1]), 5 )
print( "Le resultat est :" + str(monresultat) )
```

```
if( operateur == "*" ):
  print("ok plus")
  print("seconde ligne")
else:
  if( operateur == "-" ):
    print("moins")
```

Exercice :
Paramétrez l'opération a faire en la passant en paramètre
ex :
calcul.py 2 "*" 3

Variante générant moins de lignes de code :

```
if( operateur == "*" ):
  print("ok plus")
  print("seconde ligne")
#else:
#  if( operateur == "-" ):
elif( operateur == "-" ):
  print("moins")
```

```
#!/usr/bin/python3

import sys

position = 0
resultat = 0

for param in sys.argv:
  if( position==1 ):
    if(param=="--help"):
      print("Voici la doc de cette commande....")
    else:
      operateur=param

  if( position>1 ):
    if(operateur=="+"):
      resultat+= int(param) # equivaut a : resultat = resultat + int(param)
    if(operateur=="-"):
      resultat-= int(param)
    if(operateur=="*"):
      resultat*= int(param)
    if(operateur=="/"):
      resultat/= int(param)

    # je fais mon opération pour chaque valeur

  position+=1

if(resultat!=0):
  print("Le résultat est : "+str(resultat))
```

Mise en oeuvre de base de données pgsql en python :
https://wiki.postgresql.org/wiki/Psycopg2_Tutorial

Afficher le cumul de valeurs saisies au clavier :

```
#!/usr/bin/python3

tnombres = []
while(True):
    v = input()
    if(v==''):
        break
    tnombres.append(int(v))

cumul = 0
for n in tnombres:
    cumul += n

print("Le cumul est : "+str(cumul))
```

Requête dans postgresql en python :

```
#!/usr/bin/python3

import psycopg2

cnx = psycopg2.connect("dbname='stage' user='postgres' host='localhost'
password='Pa$$w0rd' ")
cur = cnx.cursor()
cur.execute("""SELECT * from clients""")
rows = cur.fetchall()
for row in rows:
    print( "  id: ", row[0] )
    print( "  nom: ", row[1] )
    print( "  prénom: ", row[2] )

f= open("guru99.txt","w+")
f.write("Bonjour tout le monde")
```