Session de Renforcement

Les Bases de Python

Un parcours guidé pour consolider vos fondations et repartir du bon pied en programmation

Objectifs de cette session

1

Revoir les concepts fondamentaux

Variables, types de données, opérations de base

2

Comprendre la logique

Conditions, boucles, et structuration du code

3

Maîtriser la syntaxe

Écrire du code Python correct et lisible

4

Pratiquer ensemble

Exercices guidés pour ancrer les apprentissages

5

Gagner en confiance

Se sentir prêt pour la suite du cours

Qu'est-ce que Python ?

Python est un langage de programmation créé en 1991 par Guido van Rossum. C'est l'un des langages les plus populaires au monde, et pour de bonnes raisons !

Votre première instruction Python

```
print("Bonjour le monde!")
```

Cette simple ligne affiche du texte à l'écran. Simple, n'est-ce pas ?

Pourquoi choisir Python ?

- **Syntaxe claire** : se lit presque comme de l'anglais
- **Polyvalent** : web, data science, IA, automatisation
- **Bibliothèques riches** : des outils pour tout faire
- **Grande communauté** : aide disponible partout
- **Utilisé en entreprise** : Google, Netflix, NASA...

Les variables : vos boîtes de stockage

Une variable est comme une boîte avec une étiquette où l'on range une valeur. On peut ensuite réutiliser cette valeur en appelant le nom de la boîte.

```
nom = "Alice"
age = 25
taille = 1.75
est_etudiant = True

print(nom)  # Affiche: Alice
print(age)  # Affiche: 25
```

Règles de nommage

- Commence par une lettre ou _
- Pas d'espaces ni d'accents
- Sensible à la casse : `age` ≠ `Age`

Bonnes pratiques

- Noms **explicites** : `prix_total` plutôt que `pt`
- Style **snake_case** : mots séparés par _
- Éviter les mots réservés comme print

Les types de données principaux

Textes (str)

```
prenom = "Marie"
message = 'Bonjour'
```

Les chaînes de caractères pour du texte

Nombres entiers (int)

```
age = 42
temperature = -5
```

Pour compter, indexer, calculer

Nombres décimaux (float)

```
prix = 19.99
pi = 3.14159
```

Pour les mesures et calculs précis

Booléens (bool)

```
est_majeur = True
pluie = False
```

Pour les questions vrai/faux

Listes (list)

```
notes = [12, 15, 18]
noms = ["A", "B"]
```

Collections ordonnées d'éléments

Dictionnaires (dict)

```
personne = {"nom":
"Dupont","age": 30}
```

Paires clé-valeur pour structurer

Les opérations de base

Opérations mathématiques

```
a = 10 + 5    # Addition → 15
b = 10 - 5    # Soustraction → 5
c = 10 * 5    # Multiplication → 50
d = 10 / 3    # Division → 3.333...
e = 10 // 3   # Division entière → 3
f = 10 % 3    # Modulo (reste) → 1
g = 10 ** 2   # Puissance → 100
```

Opérations sur les textes

```
texte1 = "Bon" + "jour"
# Résultat: "Bonjour"

texte2 = "Python" * 3
# Résultat: "PythonPythonPython"

texte3 = "Hello"
print(len(texte3))
# Résultat: 5
```

Astuce : Le symbole % (modulo) est très utile pour vérifier si un nombre est pair : `nombre % 2 == 0` renvoie True si le nombre est pair !

Les conditions : prendre des décisions

Les structures conditionnelles permettent à votre programme d'exécuter différentes actions selon les situations.

Structure de base

```
age = 20

if age < 18:
    print("Mineur")
elif age < 65:
    print("Adulte")
else:
    print("Senior")
```

Opérateurs de comparaison

- `==` égal à
- `!=` différent de
- `<` plus petit que
- `>` plus grand que
- `<=` plus petit ou égal
- `>=` plus grand ou égal

Vous pouvez aussi combiner plusieurs conditions avec `and` (et) et `or` (ou) :

```
if age >= 18 and permis == True:
 print("Vous pouvez conduire")
```

Les boucles : répéter des actions

Les boucles permettent d'exécuter plusieurs fois le même bloc de code sans avoir à le réécrire.

Boucle FOR

Quand on sait **combien de fois** répéter

```
# Répéter 5 fois
for i in range(5):
    print(f"Tour n°{i}")

# Parcourir une liste
noms = ["Alice", "Bob", "Charlie"]
for nom in noms:
    print(f"Bonjour {nom}")
```

Boucle WHILE

Tant qu'une **condition est vraie**

```
compteur = 0
while compteur < 3:
    print(f"Compteur: {compteur}")
    compteur += 1  # Important !

# Boucle infinie à éviter :
# while True:
#     print("Sans fin...")
```

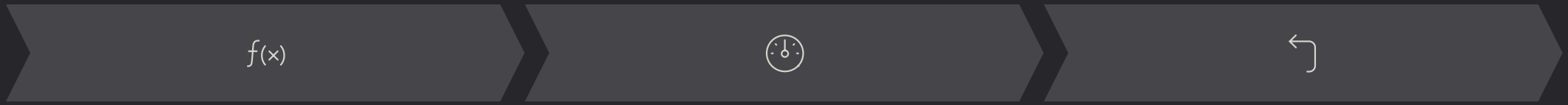
Attention : Dans une boucle while, n'oubliez jamais de modifier la variable de condition, sinon la boucle ne s'arrêtera jamais !

Les fonctions : réutiliser du code

Une fonction est un bloc de code réutilisable qui effectue une tâche précise. C'est comme créer votre propre commande personnalisée !

```
def dire_bonjour(nom):
    """Cette fonction dit bonjour à quelqu'un"""
    message = f"Bonjour {nom} !"
    return message

# Appel de la fonction
resultat = dire_bonjour("Alice")
print(resultat)  # Affiche: Bonjour Alice !
```



Définition

Mot-clé `def` suivi du nom et des paramètres entre parenthèses

Paramètres

Variables que la fonction reçoit en entrée pour travailler

Retour

Mot-clé `return` pour renvoyer un résultat au code appelant

Fonction avec paramètres optionnels

```
def calculer_prix(age, etudiant=False):
 if etudiant:
 return 7
 elif age < 12:
 return 5
 else:
 return 10

prix1 = calculer_prix(20) # 10€
prix2 = calculer_prix(20, True) # 7€
```

Les listes : collections ordonnées

Les listes sont des conteneurs qui peuvent stocker plusieurs valeurs dans un ordre précis. Elles sont modifiables et très flexibles.

Création et accès

```
fruits = ["pomme", "banane", "orange"]

# Accès par index (commence à 0)
print(fruits[0])   # "pomme"
print(fruits[-1])  # "orange" (dernier)

# Tranche (slice)
print(fruits[0:2]) # ["pomme", "banane"]
```

Modifications courantes

```
fruits = ["pomme", "banane"]

# Ajouter
fruits.append("kiwi")      # À la fin
fruits.insert(1, "poire")  # À la position 1

# Supprimer
fruits.remove("banane")    # Par valeur
fruits.pop()               # Dernier élément
fruits.pop(0)              # À la position 0
```

Opérations utiles

```
nombres = [4, 2, 7, 1, 9]
longueur = len(nombres) # 5
triee = sorted(nombres) # [1, 2, 4, 7, 9]
maximum = max(nombres) # 9
somme = sum(nombres) # 23
```

Les dictionnaires : données structurées

Un dictionnaire stocke des paires clé-valeur, comme un vrai dictionnaire associe des mots à leurs définitions.

```
personne = {
    "nom": "Dupont",
    "prenom": "Marie",
    "age": 25,
    "ville": "Paris"
}
```

Accéder aux valeurs

```
# Avec crochets
nom = personne["nom"]

# Avec .get() (plus sûr)
age = personne.get("age")
pays = personne.get("pays", "France")
```

Modifier et ajouter

```
# Modifier une valeur
personne["age"] = 26

# Ajouter une clé
personne["metier"] = "Développeuse"

# Supprimer
del personne["ville"]
```

Parcourir un dictionnaire

```
# Les clés et valeurs
for cle, valeur in personne.items():
 print(f"{cle}: {valeur}")

# Seulement les clés
for cle in personne.keys():
 print(cle)
```

Gérer les erreurs avec try/except

Les erreurs sont inévitables en programmation. Le bloc try/except permet de les anticiper et d'éviter que votre programme ne plante.

Structure de base

```
try:
    age = int(input("Votre âge ? "))
    print(f"Vous avez {age} ans")
except ValueError:
    print("Erreur : entrez un nombre")
```

Si l'utilisateur tape "vingt" au lieu de "20", le programme ne plante pas mais affiche un message d'erreur poli.

Gestion multiple

```
try:
    nombre = int(input("Un nombre : "))
    resultat = 10 / nombre
except ValueError:
    print("Ce n'est pas un nombre")
except ZeroDivisionError:
    print("Division par zéro impossible")
except Exception as e:
    print(f"Erreur inconnue : {e}")
```

Conseil : Utilisez des blocs try/except spécifiques pour chaque type d'erreur plutôt qu'un seul bloc générique. Cela vous aide à mieux comprendre et résoudre les problèmes.

Bonnes pratiques de programmation

✅ Code bien écrit

```
# Noms explicites
prix_total_ttc = 119.99
age_utilisateur = 25

# Indentation cohérente (4 espaces)
if age_utilisateur >= 18:
    print("Accès autorisé")
    acces_compte = True

# Commentaires utiles
# Applique une réduction de 10% aux étudiants
if est_etudiant:
    prix_final = prix * 0.9
```

❌ Code à éviter

```
# Noms cryptiques
pt = 119.99
a = 25

# Indentation incohérente
if a>=18:
 print("ok")
   x=True

# Commentaires inutiles
# Affecte 10 à x
x = 10
```

01

Nommage clair

Variables et fonctions avec des noms descriptifs en `snake_case`

02

Indentation uniforme

Toujours 4 espaces, jamais de tabulations

03

Commentaires pertinents

Expliquer le "pourquoi", pas le "quoi"

04

Fonctions courtes

Une fonction = une responsabilité claire

Débogage : trouver et corriger les erreurs

Apprendre à déboguer est aussi important qu'apprendre à coder. Voici les erreurs les plus fréquentes et comment les résoudre.

SyntaxError

Cause : Faute de frappe, parenthèse ou guillemet manquant

```
# Mauvais
print("Bonjour"
# Bon
print("Bonjour")
```

IndentationError

Cause : Mauvaise indentation du code

```
# Mauvais
if age > 18:
print("Majeur")
# Bon
if age > 18:
    print("Majeur")
```

NameError

Cause : Variable utilisée avant d'être définie

```
# Mauvais
print(nom)
# Bon
nom = "Alice"
print(nom)
```

TypeError

Cause : Opération impossible avec ce type

```
# Mauvais
resultat = "5" + 10
# Bon
resultat = int("5") + 10
```

IndexError

Cause : Index hors de la liste

```
# Mauvais
liste = [1, 2, 3]
print(liste[5])
# Bon
print(liste[2])
```

Techniques de débogage efficaces

- Utilisez `print()` pour afficher les valeurs des variables à chaque étape
- Lisez attentivement les messages d'erreur : ils indiquent la ligne et le type de problème
- Testez votre code par petits morceaux plutôt que tout d'un coup
- Commentez des parties du code pour isoler le problème

Méthodologie de résolution de problèmes

Face à un exercice ou un projet, suivez cette démarche structurée pour ne pas vous perdre.

Comprendre le problème

Lisez l'énoncé plusieurs fois. Identifiez clairement les entrées (ce que vous recevez) et les sorties (ce que vous devez produire). Reformulez avec vos propres mots.

Décomposer en étapes

Divisez le grand problème en petites tâches simples. Écrivez ces étapes en français avant de coder. Par exemple : "1. Demander l'âge, 2. Vérifier si majeur, 3. Afficher le résultat".

Résoudre étape par étape

Commencez par la partie la plus simple. Écrivez le code pour une seule étape, testez-la, puis passez à la suivante. Ne tentez pas tout d'un coup.

Tester régulièrement

Après chaque ajout de code, testez avec différentes valeurs, y compris des cas limites (nombres négatifs, texte vide, etc.). Vérifiez que tout fonctionne avant de continuer.

Améliorer et optimiser

Une fois que ça marche, relisez votre code. Peut-on le rendre plus clair ? Plus court ? Ajouter des commentaires ? C'est le moment de peaufiner.

Ressources pour progresser

Pratiquez quotidiennement

Même 15 minutes par jour font la différence. Créez de petits programmes sur des sujets qui vous intéressent : calculatrices, jeux simples, automatisation de tâches répétitives.

Documentation officielle

Le site **docs.python.org** contient toutes les informations sur Python. Habituez-vous à consulter la documentation, c'est une compétence essentielle pour tout développeur.

Petits projets personnels

Créez des outils qui vous sont utiles : gestionnaire de budget, tracker d'habitudes, convertisseur d'unités. L'apprentissage par projet est le plus efficace.

Demandez de l'aide

Ne restez jamais bloqué trop longtemps. Posez des questions à vos professeurs, camarades, ou sur les forums. Expliquer votre problème aide souvent à trouver la solution !

Ne jamais abandonner

> "La programmation, c'est comme apprendre une langue étrangère. Au début, tout semble compliqué et étranger. Mais avec de la pratique régulière, ça devient naturel et même amusant. Les erreurs ne sont pas des échecs, ce sont des opportunités d'apprendre."

Les erreurs sont normales

Même les développeurs expérimentés en font quotidiennement. C'est en corrigeant vos erreurs que vous progressez le plus vite.

La patience est clé

Rome ne s'est pas faite en un jour. Donnez-vous le temps d'assimiler les concepts. La compréhension viendra avec la pratique.

Célébrez les petites victoires

Chaque programme qui fonctionne est une victoire. Soyez fier de vos progrès, aussi petits soient-ils.