**Résumé de la section**

Il est très courant que nous ayons besoin de gérer des dates en programmation.

Que ce soit pour gérer les dates d'anniversaires des clients d'un site web, annoncer un compte à rebours ou encore calculer le nombre de jours entre deux dates.

Pour réaliser ces opérations, Python dispose de la bibliothèque native **datetime** qui permet de gérer des objets représentant des dates et des heures.

**Comment un ordinateur calcule-t-il le temps ?**

Vous avez peut-être déjà entendu parler de « l'Epoch », une date arbitraire aussi appelée « heure Unix » :

L'heure Unix ou heure Posix (aussi appelée Unix Timestamp) est une mesure du temps basée sur le nombre de secondes écoulées depuis le 1er janvier 1970 00:00:00 UTC, hors secondes intercalaires. Elle est utilisée principalement dans les systèmes qui respectent la norme POSIX1, dont les systèmes de type Unix, d'où son nom. C'est la représentation POSIX du temps.

*définition de*[***Wikipedia***](https://fr.wikipedia.org/wiki/Heure_Unix)

Le calcul du temps peut s'avérer complexe pour tout un tas de raison comme les [**secondes intercalaires**](https://fr.wikipedia.org/wiki/Seconde_intercalaire), [**les fuseaux horaires**](https://fr.wikipedia.org/wiki/Fuseau_horaire) ou encore [**l'heure d'été**](https://fr.wikipedia.org/wiki/Heure_d'%C3%A9t%C3%A9).

Généralement cela ne pose pas de problèmes (on a rarement besoin de prendre en compte les potentielles microsecondes de différence entre le temps universel et le temps solaire).

Mais nous verrons dans la suite de cet article que notamment les fuseaux horaires doivent être pris en compte pour éviter des erreurs qui peuvent s'avérer problématiques (comme envoyer un mail à 3h du matin au lieu de 9).

Avec Python, il est possible d'afficher le nombre de secondes écoulées depuis l'heure Unix avec la fonction **time** du module **time** :

1. >>> from time import time
2. >>> time()
3. 1634911800.733942

Au moment où j'écris cet article, il s'est donc écoulé 1634911800 secondes depuis le 1er janvier 1970.

**Représenter le temps avec datetime**

Pour aller un peu plus loin et manipuler des dates (et des heures), Python dispose du module **datetime** ainsi que trois classes principales :

* **date**
* **time**
* **datetime**

Vous vous en doutez, la classe **date** sert à gérer des dates, **time** à gérer du temps et **datetime** les deux.

Toutes ces classes disposent d'attributs permettant d'indiquer une année, un mois, un jour, une heure et également des informations de fuseau horaire.

Voyons dans le détail comment créer ces objets :

1. >>> from datetime import date, time, datetime
2. >>> date(year=2021, month=10, day=22)
3. datetime.date(2021, 10, 22)
4. >>> time(hour=10, minute=19, second=10)
5. datetime.time(10, 19, 10)
6. >>> datetime(year=2021, month=10, day=22, hour=10, minute=19, second=10)
7. datetime.datetime(2021, 10, 22, 10, 19, 10)

Le nom du module **datetime** étant le même que le nom de la classe, on confond souvent les deux. Si vous importez le module directement, il faudra donc utiliser **datetime.datetime** ce qui est assez confu. Je préfère ainsi importer directement la classe avec **from datetime import datetime**.

Remarquez que tous les noms des paramètres de ces classes sont au singulier (hour, day, minute...). Ce ne sera pas toujours le cas dans les différentes classes et fonctions que nous allons voir dans cet article, il est donc important de noter quand c'est le cas ou non.

Ces trois classes nous permettent de représenter des périodes dans le temps en utilisant des nombres entiers pour les années, mois, jours, etc.

Les classes **datetime** et **date** disposent également de deux fonctions, **now** et **today** qui permettent respectivement de récupérer la date et l'heure d'aujourd'hui (pour **datetime**) et la date d'aujourd'hui (pour **date**) :

1. >>> datetime.now()
2. datetime.datetime(2021, 10, 22, 10, 25, 7, 742193)
3. >>> date.today()
4. datetime.date(2021, 10, 22)

La fonction **today** est disponible sur les objets **datetime** mais la fonction **now** n'est pas disponible sur les objets **date**.

1. >>> datetime.today()
2. datetime.datetime(2021, 10, 22, 10, 26, 36, 718675)
3. >>> datetime.now()
4. datetime.datetime(2021, 10, 22, 10, 26, 39, 182568)
5. >>> date.now()
6. Traceback (most recent call last):
7. File "<stdin>", line 1, in <module>
8. AttributeError: type object 'datetime.date' has no attribute 'now'

Une fois que nous avons créé un objet de time **date**, **time** ou **datetime**, nous pouvons récupérer les différents attributs de ces objets :

1. >>> today = date.today()
2. >>> today.day
3. 22

Là encore, notez l'utilisation du singulier pour le nom de l'attribut.

À noter également qu'il est possible de récupérer les valeurs de ces attributs mais pas de les modifier directement.

1. >>> today = date.today()
2. >>> today.day = 23
3. Traceback (most recent call last):
4. File "<stdin>", line 1, in <module>
5. AttributeError: attribute 'day' of 'datetime.date' objects is not writable

Pour remplacer un élément d'un objet **datetime** on peut utiliser la méthode **replace** :

1. >>> today = date.today()
2. >>> tomorrow = today.replace(day=today.day + 1)
3. >>> today
4. datetime.date(2021, 10, 22)
5. >>> tomorrow
6. datetime.date(2021, 10, 23)

Je vous déconseille cependant d'utiliser cette méthode car elle ne prend pas en compte les problèmes de fuseaux horaires et du passage à l'heure d'été. Nous verrons plus loin dans cet article comment modifier une date de la bonne façon.

**Créer une date avec une chaîne de caractères**

Il arrive souvent que nous souhaitions créer un objet **datetime** non pas à partir de nombres entiers mais directement à partir d'une chaîne de caractères.

Le problème qui se pose est que la représentation des dates sous forme de chaîne de caractères peut prendre beaucoup de formes.

Par exemple aux États-Unis, les dates sont généralement représentées sous le format MM-DD-YYYY (par exemple 10-22-2021 pour représenter le 22 octobre 2021).

En Europe on utilise plutôt DD-MM-YYYY (22-10-2021 pour le 22 octobre 2021).

Tout au long de cet article, je représenterai les jours, mois et années avec les lettres **D**, **M** et **Y** (**D**ay, **M**onth et **Y**ear).

La norme [**ISO8601**](https://en.wikipedia.org/wiki/ISO_8601) a été créée pour pallier à ce problème. Selon cette norme, on va de l'élément le plus grand au plus petit, une date serait donc représentée par **YYYY-MM-DD HH:MM:SS**.

Les objets de la classe **datetime** permettent de créer une instance à partir de ce format grâce à la fonction **fromisoformat** :

1. >>> from datetime import date
2. >>> date.fromisoformat("2021-10-22")
3. datetime.date(2021, 10, 22)

Dans le cas où une date n'est pas dans le format ISO8601 (c'est souvent le cas), on peut utiliser à la place la fonction **strptime** qui permet de spécifier le formatage de la date.

Imaginons une date représentée par la chaîne de caractères suivante :

1. '22 Oct 2021'

Grâce à **strptime** et à un langage spécifique dont vous pouvez retrouver les caractéristiques sur [**ce site**](https://strftime.org/), on peut indiquer le format de la date en le passant en 2e argument à la fonction :

1. >>> datetime.strptime("22 Oct 2021", "%d %b %Y")
2. datetime.datetime(2021, 10, 22, 0, 0)

Cette méthode n'est disponible que sur la classe **datetime**. Si vous n'arrivez pas à vous souvenir du nom de cette méthode, sachez que le **p** de **strptime** signifie **parser** (« analyseur » en français). **strptime** est donc un raccourci de **str**ing **p**arser **time**.

Et si nous souhaitons réaliser l'opération inverse, nous pouvons le faire avec la fonction **strftime** (**str**ing **f**rom **time**) :

1. >>> now = datetime.now()
2. >>> now.strftime("%d %b %Y")
3. '22 Oct 2021'

Deux modules non inclus dans la bibliothèque standard de Python permettent de manipuler les dates avec beaucoup plus de facilité :

* [**dateutil**](https://pypi.org/project/python-dateutil/)
* [**dateparser**](https://pypi.org/project/dateparser/)

**dateutil** nous permet grâce au module **parser** et la fonction **parse** de créer un objet **datetime** sans avoir besoin de spécifier le format comme nous l'avons fait avec **strptime** et avec beaucoup plus de libertés :

1. >>> from dateutil import parser
2. >>> parser.parse("12 october 2021 at 9 am and 18 minutes")
3. datetime.datetime(2021, 10, 12, 9, 18)

Notez cependant que nous utilisons ici **october** au lieu de **octobre**, cette bibliothèque ne comprenant pas le français :

1. >>> parser.parse("12 octobre 2021")
2. File "<stdin>", line 1, in <module>
3. raise ParserError("Unknown string format: %s", timestr)
4. dateutil.parser.\_parser.ParserError: Unknown string format: 12 octobre 2021

La bibliothèque **dateparser** elle va cependant beaucoup plus loin en nous permettant d'utiliser des mots marqueurs de temps comme « aujourd'hui » ou « demain » et ce même dans différentes langues (🤯) :

1. >>> dateparser.parse("aujourd'hui")
2. datetime.datetime(2021, 10, 22, 11, 11, 18, 244967)
3. >>> dateparser.parse("demain")
4. datetime.datetime(2021, 10, 23, 11, 10, 19, 246512)
5. >>> dateparser.parse("ontem") # La date d'hier en Portugais
6. datetime.datetime(2021, 10, 21, 11, 10, 46, 885066)

Cette bibliothèque est vraiment incroyable et j'ai encore du mal à trouver ses limites :

1. >>> dateparser.parse("Il y a un mois")
2. datetime.datetime(2021, 9, 22, 11, 12, 42, 170722)
3. >>> dateparser.parse("One year ago at midnight")
4. datetime.datetime(2020, 10, 22, 0, 0)

**La gestion des fuseaux horaires**

La gestion des fuseaux horaires est possible de différentes façons, surtout depuis l'ajout dans la version 3.9 de Python d'un nouveau module **zoneinfo**.

Les différents modules que nous allons voir dans cette partie utilisent une base de données appelée [***tz database***](https://fr.wikipedia.org/wiki/Tz_database) également appelée *IANA database*.

Vous pouvez retrouver tous les fuseaux horaires de cette base de données ainsi que leur nom sur [**la page wikipedia associée**](https://en.wikipedia.org/wiki/List_of_tz_database_time_zones).

Les dates « naive » et « aware »

Une date « naive » est une date qui n'a aucune information de fuseau horaire (*timezone* en anglais). À l'inverse une date « aware » est définie selon un fuseau horaire précis.

Par défaut, les objets **datetime** comme ceux que nous avons créés ci-dessus sont « naive ». La façon la plus simple de vérifier les informations de fuseau horaire d'un objet **datetime** est de vérifier l'attribut **tzinfo** :

1. >>> from datetime import datetime
2. >>> now = datetime.now()
3. >>> now.tzinfo
4. None

Ajouter un fuseau horaire spécifique

Il existe différentes façons d'ajouter des informations de fuseau horaire à un objet **datetime** en fonction des versions Python que vous utilisez.

Depuis la version 3.9 de Python, le module de la bibliothèque standard **zoneinfo** permet de gérer les fuseaux horaires. Si vous utilisez une version de Python inférieure à 3.9, je vous conseille de regarder des modules comme [**dateutil**](https://pypi.org/project/python-dateutil/) ou [**pytz**](https://pypi.org/project/pytz/).

Les principes que nous allons voir dans la suite de cet article restent les mêmes peu importe le module utilisé.

Grâce à la classe **ZoneInfo** du module **zoneinfo** on peut créer un fuseau horaire en nous basant sur la [**base de données IANA**](https://en.wikipedia.org/wiki/List_of_tz_database_time_zones).

Si je souhaite récupérer le fuseau horaire de Vancouver, je peux utiliser la chaîne de caractères **"America/Vancouver"** :

1. >>> from zoneinfo import ZoneInfo
2. >>> now\_in\_vancouver = datetime.now(tz=ZoneInfo("America/Vancouver"))
3. >>> now\_in\_montreal = datetime.now(tz=ZoneInfo("America/Montreal"))
4. >>> now\_in\_montreal.hour
5. 12
6. >>> now\_in\_vancouver.hour
7. 9

On retrouve bien les 3 heures de décalages entre Montréal et Vancouver.

Étant à Montréal à l'heure où j'écris cet article, j'aurais pu utiliser **datetime.now()** sans préciser le paramètre **tz** pour récupérer la date et l'heure de l'endroit où je me trouve.

J'aurais donc eu un objet « aware » (**now\_in\_vancouver**) et un objet « naive » (**now\_in\_montreal**).

Cela ne pose pas de problème tant que nous n'effectuons aucune opération entre ces deux objets. En cas de comparaison ou de soustraction par exemple, nous avons une erreur si nous utilisons un objet « naive » et un objet « aware » :

1. >>> now\_in\_vancouver = datetime.now(tz=ZoneInfo("America/Vancouver"))
2. >>> now\_in\_montreal = datetime.now()
3. >>> now\_in\_vancouver > now\_in\_montreal
4. File "<stdin>", line 1, in <module>
5. TypeError: cant compare offset-naive and offset-aware datetimes
6. >>> now\_in\_vancouver - now\_in\_montreal
7. File "<stdin>", line 1, in <module>
8. TypeError: cant subtract offset-naive and offset-aware datetimes

Pour changer de fuseau horaire, on utilise la méthode **astimezone** :

1. >>> now\_in\_paris = now\_in\_vancouver.astimezone(ZoneInfo("Europe/Paris"))
2. >>> print(now\_in\_paris.hour)
3. 18

Là encore, la différence est cohérente (9h de décalage entre Vancouver et Paris, on passe donc de 9h à 18h).

Le Temps universel coordonné (UTC)

Vous avez déjà probablement entendu les acronymes UTC ou GMT, souvent utilisés quand vous devez planifier une réunion avec un collègue à l'autre bout du monde.

On fait souvent la confusion entre UTC et un fuseau horaire.

UTC n'est pas un fuseau horaire, c'est une échelle de temps qui a été acceptée par la grande majorité des pays comme échelle de temps universelle.

**Petite anecdote intéressante** : l'appellation correcte en anglais du temps universel coordonné (TUC) serait coordinated universal time, abrégé en CUT. Les experts de l’Union internationale des télécommunications étaient d’accord pour définir une abréviation commune à toutes les langues, mais ils étaient divisés sur le choix de la langue entre le français et l'anglais. Finalement, c’est le compromis UTC, nécessitant un effort des deux parties, qui fut choisi. C’est cette notation qui est utilisée par la norme ISO 8601. (de [**wikipedia**](https://fr.wikipedia.org/wiki/Temps_universel_coordonn%C3%A9))

On pense souvent à tort qu'UTC correspond à un fuseau horaire car nous pouvons convertir un object **datetime** en UTC grâce au module **timezone** de la bibliothèque **datetime** en le passant au paramètre **tz** :

1. >>> from datetime import datetime, timezone
2. >>> now = datetime.now(tz=timezone.utc)
3. >>> now
4. datetime.datetime(2021, 10, 22, 15, 57, 4, 822209, tzinfo=datetime.timezone.utc)
5. >>> now.tzinfo
6. datetime.timezone.utc

Quand vous comparez des dates ou que vous souhaitez les modifier, il est très important de toujours le faire sur une base UTC. Je répète : **ne faites pas d'opérations sur des dates avec un fuseau horaire spécifique**.

Il est notamment dangereux d'effectuer une opération sur une date avec un fuseau horaire précis car les changements d'heures été / hiver ne seront pas pris en compte.

Par exemple à Montréal, il existe deux fuseaux horaires, EDT et EST (Eastern Daylight Time et Estern Standard Time), selon les périodes de l'année.

En 2020, le passage de l'heure d'hiver à l'heure d'été a eu lieu dans la nuit du 7 au 8 mars.

1. >>> from datetime import datetime
2. >>> from zoneinfo import ZoneInfo
4. >>> montreal\_tz = ZoneInfo('America/Montreal')
5. >>> print(datetime(2020, 3, 7, 13, 0, 0, tzinfo=montreal\_tz))
6. 2020-03-07 13:00:00-05:00
7. >>> print(datetime(2020, 3, 8, 13, 0, 0, tzinfo=montreal\_tz))
8. 2020-03-08 13:00:00-04:00

Dans le code ci-dessus, on peut voir la différence avec le temps UTC à la fin de l'affichage des dates (2020-03-07 13:00:00👉**-05:00**👈)

Le 7 mars, on a une différence de 5h entre le fuseau horaire de Montréal et UTC. Le 8 mars, la différence n'est plus que de 4h entre le fuseau horaire de Montréal et UTC.

On peut également voir cette différence avec le changement de nom du fuseau horaire :

1. >>> montreal\_tz = ZoneInfo('America/Montreal')
2. >>> print(datetime(2020, 3, 7, 13, 0, 0, tzinfo=montreal\_tz).tzname())
3. EST
4. >>> print(datetime(2020, 3, 8, 13, 0, 0, tzinfo=montreal\_tz).tzname())
5. EDT

La différence d'une heure par rapport à UTC est normale car on a avancé d'une heure dans la nuit du 7 au 8 mars 2020 à Montréal. L'écart relatif par rapport à UTC n'est donc plus le même entre le 7 et le 8 mars.

L'heure universelle UTC elle n'a pas bougé, mais l'heure à Montréal oui. Et ces changements n'arrivent pas toujours au même moment partout dans le monde.

C'est pour cette raison que tous les ans, pendant 3 semaines environ, il n'y a plus que 5h de décalage entre Paris et Montréal au lieu de 6 (car le passage à l'heure d'été en France se fait en général 2 à 3 semaines après le changement au Canada).

**Et cette différence peut causer des problèmes lorsque vous réalisez des opérations sur une date.**

Pour être très précis, EST et EDT ne sont pas des fuseaux horaires en soi. Un fuseau horaire est défini par un continent et le plus souvent une ville, comme nous l'avons vu dans la base de données IANA (par exemple Europe/Paris ou America/Montreal).

EST et EDT représentent la différence de temps avec UTC. Quand nous indiquons un fuseau horaire à **ZoneInfo**, nous donnons donc le nom du fuseau horaire (par exemple **America/Montreal**) et en fonction de la période de l'année, nous nous trouverons donc soit en EST soit en EDT, avec une différence par rapport à UTC qui ne sera pas la même.

D'ailleurs, pour afficher le ou les noms associés au fuseau horaire où l'on se trouve on peut utiliser le module **time** :

1. >>> import time
2. >>> time.tzname
3. ('EST', 'EDT')

La mauvaise façon de faire

Voyons maintenant un exemple dans lequel nous travaillons directement avec un objet **datetime** sur le fuseau horaire de Montréal.

Nous souhaitons ajouter un jour à la date du 7 mars (nous utilisons pour ce faire la classe **timedelta** que nous verrons plus en détail dans la suite de cet article) :

1. >>> from datetime import datetime, timedelta
2. >>> from zoneinfo import ZoneInfo
3. >>> tz\_montreal = ZoneInfo('America/Montreal')
4. >>> march\_7 = datetime(2020, 3, 7, 13, 0, 0, tzinfo=tz\_montreal)
5. >>> march\_8 = march\_7 + timedelta(days=1)
6. >>> print(march\_8)
7. 2020-03-08 13:00:00-04:00

Remarquez que l'heure n'a pas bougé. On vient juste d'ajouter 1 jour à notre date, sans prendre en compte le passage à l'heure d'été.

La bonne façon de faire

Reprenons maintenant le même exemple en convertissant d'abord notre date en UTC avant de faire l'addition, puis en modifiant le fuseau horaire par la suite :

1. >>> from datetime import datetime, timedelta
2. >>> from zoneinfo import ZoneInfo
4. >>> march\_7 = datetime(2020, 3, 7, 13, 0, 0)
5. >>> march\_7\_utc = march\_7.astimezone(ZoneInfo("UTC"))
7. >>> march\_8 = march\_7\_utc + timedelta(days=1)
8. >>> march\_8 = march\_8.astimezone(montreal\_tz)
9. >>> print(march\_8)
10. 2020-03-08 14:00:00-04:00

Nous remarquons cette fois-ci que l'heure d'été a bien été prise en compte et notre date affiche 14h au lieu de 13.

**En faisant l'opération sur une date basée sur UTC puis en convertissant par la suite dans le fuseau horaire voulu, nous obtenons le bon résultat.**

Prenons un autre cas de figure pour illustrer les problèmes causés lors du calcul de la différence entre deux dates :

1. >>> from datetime import datetime
2. >>> from zoneinfo import ZoneInfo
4. >>> utc = ZoneInfo("UTC")
5. >>> montreal\_tz = ZoneInfo('America/Montreal')
7. >>> march\_7 = datetime(2020, 3, 7, 13, 0, 0, tzinfo=montreal\_tz)
8. >>> march\_10 = datetime(2020, 3, 10, 13, 0, 0, tzinfo=montreal\_tz)
9. >>> march\_7\_utc = march\_7.astimezone(utc)
10. >>> march\_10\_utc = march\_10.astimezone(utc)
12. >>> print(march\_10 - march\_7)
13. 3 days, 0:00:00
14. >>> print(march\_10\_utc - march\_7\_utc)
15. 2 days, 23:00:00

Dans le premier cas de figure (**march\_10 - march\_7**), une simple soustraction est effectuée entre les deux dates au fuseau horaire de Montréal. On obtient donc une différence parfaite de 3 jours qui ne prend là encore pas en compte le passage à l'heure d'été.

Dans le deuxième cas de figure, avec les dates converties en UTC, on observe cette fois-ci bien une différence d'1h avec le résultat précédant, indiquant la prise en compte de l'heure qui a été "perdue" lors du passage à l'heure d'été.

**Calcul sur un objet datetime**

Nous l'avons vu brièvement dans la partie précédente avec les **timedelta**, il est possible de réaliser des opérations entre deux dates ou de modifier une date.

La classe **timedelta** permet d'ajouter un intervalle de temps à une date. Attention, **timedelta** ne représente pas une date mais un intervalle. Vous remarquerez d'ailleurs que cette fois-ci, les attributs sont au pluriel :

1. >>> from datetime import timedelta
2. >>> timedelta(days=20)
3. datetime.timedelta(days=20)

La classe **timedelta** permet d'ajouter ou de soustraire des jours, heures, secondes et même microsecondes. On peut sans problème utiliser des nombres positifs et négatifs en même temps.

Dans l'exemple ci-dessous, nous ajoutons 15 jours et soustrayons 5 heures à la date actuelle :

1. >>> now = datetime.now()
2. >>> now\_in\_15\_days\_minus\_5\_hours = now + timedelta(days=15, hours=-5)
3. >>> print(now)
4. 2021-10-22 17:43:23.840561
5. >>> print(now\_in\_10\_days\_minus\_5\_hours)
6. 2021-11-06 12:43:23.840561

La date passe logiquement au mois suivant. C'est valable également pour des mois comme le mois de février qui ne contient que 28 jours.

1. >>> feb\_27\_2022 = datetime(2022, 2, 27) # 27 février
2. >>> print(feb\_27\_2022 + timedelta(days=3)) # On ajoute 3 jours
3. 2022-03-02 00:00:00

On tombe bien sur le 2 mars et non le 30 février.

La classe **timedelta** ne permet d'ajouter que des jours, heures, minutes mais pas des mois ou des années. C'est cependant possible avec la classe **relativedelta** du module **dateutil** que nous avons vu précédemment mais qui n'est pas disponible dans la bibliothèque standard de Python :

1. >>> from dateutil.relativedelta import relativedelta
2. >>> from datetime import datetime
3. >>> now = datetime.now()
4. >>> now\_in\_2\_months = now + relativedelta(months=2)
5. >>> print(now)
6. 2021-10-22 17:52:20.878397
7. >>> print(now\_in\_2\_months)
8. 2021-12-22 17:52:20.878397

**Conclusion**

J'espère qu'avec cette partie vous y verrez un peu plus clair avec le module **datetime**. La base est assez simple à maîtriser mais il est très important dans un contexte professionnel de porter attention à toutes les subtilités que nous avons vues dans cet article.

On peut penser que quelques heures de décalage ne sont pas très graves, mais ça peut vite devenir problématique quand vous souhaitez l'anniversaire à vos clients un jour trop tôt ou que vous envoyez un mail sur le mauvais fuseau horaire.

Pour un projet perso ça peut passer, pour un projet avec des clients, il est très important de porter attention à ces détails.