Python datetime: Lidando com datas e horários | Alura Cursos Online

Vamos a um exemplo concreto de **datas** e **horários** para poder programar em Python e com o módulo `datetime`: Uma empresa nos contratou para implementar o sistema de pontos deles, controlando quando um funcionário chega e sai. O sistema deve exibir a data e a hora a cada registro, como confirmação para o funcionário.

date do datetime

Conhecemos o módulo **datetime** da biblioteca nativa do **Python**, então até sabemos pegar a data atual através da classe **date**, basta a importarmos e chamarmos o método **`today()`**:

```
from datetime import date

data_atual = date.today()
print(data_atual)
```

Como esperado:

```
2018-03-01
```

Mas calma... como esperado? Seria legal se conseguíssemos imprimir a data no formato brasileiro **`DD/MM/AAAA`** para evitar confusões! O problema é que o **`date`** automaticamente força o padrão ANSI sempre que tentamos imprimir.

Formatando nossa data em uma string

Como a classe `date` consegue nos fornecer separadamente cada seção da data, podemos resolver esse problema com uma simples **formatação de string**:

```
data_em_texto = '{}/{}/{}'.format(data_atual.day, data_atual.month,
data_atual.year)
```

Que resulta em:

```
1/3/2018
```

Já melhorou bastante, mas ainda não é exatamente o que queríamos. Repare que tanto o dia como o mês estão sem o prefixo **0**, saindo do padrão pretendido. Nesse caso poderíamos até contornar isso simplesmente adicionando um **0** antes:

```
data_em_texto = '0{}/0{}/{}'.format(data_atual.day, data_atual.month,
data_atual.year)
```

O que daria certo, mas traria problemas caso o dia ou o mês fossem maiores ou iguais a 10:

```
010/010/2018
```

São problemas e funcionalidades comuns que passamos durante nossos cursos da formação Python para web na Alura!

Formatando datas em strings usando o método strftime()

Para evitar maiores complicações, a classe `date` soluciona isso com um único método - o **`strftime()`**, que toma como parâmetro a formatação que queremos em nossa string de data e, desse modo, nos dá maior liberdade para decidirmos como queremos exibir a data.

Esta formatação usa códigos melhor explicados na documentação. Ao final do texto, também damos uma explicação breve neles. No nosso caso fica assim:

```
data_em_texto = data_atual.strftime('%d/%m/%Y')
print(data_em_texto)
```

E agora sim:

```
01/03/2018
```

Ou, no caso do nosso exemplo que resultou em `010/010/2018`:

```
10/10/2018
```

Agora só precisamos dar um jeito de armazenar o horário também. Quem será que pode cuidar disso? Como você deve ter imaginado, o mesmo módulo `datetime` do qual importamos a classe `date` também possui classes que facilitam a manipulação de horários.

O tipo datetime para cuidar de datas e horários juntos

Enquanto podemos sim usar o tipo **time**, destinado exclusivamente para horários, o módulo nos dá uma solução muito mais apropriada para o nosso problema com o tipo **datetime** - **sim, tem o mesmo nome do módulo, cuidado com a confusão!**

Uma das vantagens da classe `datetime` é que ela consegue cuidar da data e do horário ao mesmo tempo. A única diferença em nosso uso é que, em vez do método `today()`, usaremos o método **`now()`**:

```
from datetime import datetime

data_e_hora_atuais = datetime.now()
data_e_hora_em_texto = data_e_hora_atuais.strftime('%d/%m/%Y')

print(data_e_hora_em_texto)
```

O resultado é como o anterior:

```
01/03/2018
```

Opa! Apesar de já estarmos usando a classe `datetime`, que incorpora o horário, precisamos declarar na formatação que passamos como parâmetro para o `strftime()` que queremos mostrar a hora e o minuto, também:

```
data_e_hora_em_texto = data_e_hora_atuais.strftime('%d/%m/%Y %H:%M')

print(data_e_hora_em_texto)
```

e agora sim:

```
01/03/2018 12:30
```

Perfeito! Até agora aprendemos a pegar a data atual com a classe `date`, `datetime` e até aprendemos a **formatar datas**, transformando-as em strings. Mas e se precisássemos fazer o caminho contrário?

Convertendo uma string em datetime

Se tivéssemos uma string de data e quiséssemos transformar em `datetime`, o que faríamos?Novamente, um simples método resolve tudo, dessa vez o **strptime()**, da própria classe `datetime`:

```
from datetime import datetime

data_e_hora_em_texto = '01/03/2018 12:30'
data_e_hora = datetime.strptime(data_em_texto, '%d/%m/%Y %H:%M')
```

E tudo certo! Entretanto, o chefe da empresa foi testar o programa em outros computadores e alguns imprimiam horários diferentes. Por quê?

O problema do fuso horário

Chequei as configurações de um desses computadores e descobri que o relógio dele estava com um fuso horário diferente do daqui de São Paulo, onde está a empresa.

Não podemos deixar que o **tempo no nosso programa dependa de cada máquina**, porque não temos como garantir que todas as máquinas que rodarem esse programa estarão com o fuso horário que queremos. O ideal, então, seria forçar o fuso horário de São Paulo.

Fuso horário com a classe timezone

A partir do **Python 3**, temos a classe **timezone**, também do módulo `datetime`:

```
from datetime import datetime, timezone

data_e_hora_atuais = datetime.now()
fuso_horario = timezone()
print(fuso_horario)
```

Vamos ver o que é impresso na tela:

```
Traceback (most recent call last):
  File "teste.py", line 4, in >
```

```
    fuso_horario = timezone()
TypeError: Required argument 'offset' (pos 1) not found
```

A exceção **`TypeError`** que recebemos indica que o argumento **`offset`**, esperado no construtor `timezone`, não foi encontrado. Realmente, não colocamos esse argumento. Mas o que ele significa?

O parâmetro `offset` representa a diferença entre o fuso horário que queremos criar e o Tempo Universal Coordenado (UTC). No nosso caso, em São Paulo, temos uma diferença de -3 horas, mais conhecida como **UTC-3**. Sabendo disso, vamos tentar novamente:

```
fuso_horario = timezone(-3)
print(fuso_horario)
```

Agora que configuramos o parâmetro `offset`, vamos ver o que aparece na tela:

```
Traceback (most recent call last):
  File "teste.py", line 4, in <module>
    fuso_horario = timezone(-3)
TypeError: timezone() argument 1 must be datetime.timedelta, not int
```

Dessa vez a mensagem da exceção é diferente, indicando que o argumento passado para o construtor `timezone` deve ser do tipo **`datetime.timedelta`**, não um número inteiro, que foi o que passamos.

Calcular a diferença de horários com a classe timedelta

A classe **timedelta** tem a finalidade de representar uma duração e, no nosso caso, uma diferença entre horários. Vamos, agora, instanciá-la em uma variável e tentar imprimir, para ver o que acontece:

```
diferenca = timedelta()
print(diferenca)
```

Olha o que apareceu:

```
0:00:00
```

Tudo bem, 0 dias, 0 horas e 0 minutos. Mas precisamos que nosso objeto `timedelta` corresponda à diferença do **UTC**, as -3 horas:

```
diferenca = timedelta(-3)
print(diferenca)
```

E agora, o que aparece?

```
-3 days, 0:00:00
```

Mas o quê? -3 dias? A gente queria -3 horas! O problema é que o construtor `timedelta` recebe vários outros argumentos além da hora, nessa ordem:

- ***days*** (dias)
- ***seconds*** (segundos)
- ***microseconds*** (microsegundos)

- ***milliseconds*** (milisegundos)
- ***minutes*** (minutos)
- ***hours*** (horas)
- ***weeks*** (semanas)

Então se a gente só manda um **-3** para ele, esse número é interpretado como se fosse em dias. Podemos passar **0** para os primeiros 5 parâmetros e **-3** para as horas, mas isso é um pouco estranho, considerando que, de fato, **queremos definir apenas as horas**.

Usando a funcionalidade que o Python tem dos **parâmetros nomeados**, é possível especificar que estamos definindo o parâmetro de horas (*hours*), da seguinte forma:

```
diferenca = timedelta(hours=-3)
print(diferenca)
```

E dessa vez:

```
-1 day, 21:00:00
```

Ué, se a gente colocou `-3`, por que apareceu tudo isso? Acontece que o `timedelta` entende **-3 horas** como **0 dias, 0 horas e 0 minutos - 3 horas**, ou seja, **-1 dia,21 horas**.

Resolvendo o problema do fuso horário

Vamos, agora, criar um objeto timezone correspondente ao **UTC-3**, indicando essa diferença do **UTC** como parâmetro do construtor:

```
fuso_horario = timezone(diferenca)
print(fuso_horario)
```

Temos justamente o que queríamos:

```
UTC-03:00
```

Finalmente, podemos converter o tempo da máquina para o de São Paulo, usando o método **`astimezone()`**:

```
data_e_hora_sao_paulo = data_e_hora_atuais.astimezone(fuso_horario)
data_e_hora_sao_paulo_em_texto = data_e_hora_sao_paulo.strftime('%d/%m/%Y %H:%M')

print(data_e_hora_sao_paulo_em_texto)
```

Agora está tudo padronizado:

```
01/03/2018 12:30
```

Temos tudo resolvido com o fuso horário agora, mas e se fossemos mostrar nosso código para outro programador, ele entenderia o que significa o `-3`? Não fica muito fácil, né?

Aliás, para todo fuso horário teríamos que pesquisar qual a sua diferença do UTC, o que é um problema chato. Será que não há uma maneira mais simples e elegante para resolver essa questão?

Resolvendo o problema dos fusos horários com o pytz

A comunidade Python, frente a essa necessidade, criou diversas bibliotecas para facilitar a manipulação de *timezones*, como a **pytz**. Para instalar a `pytz`, você pode usar o **pip** pelo terminal:

```
pip install pytz
```

> A instalação em sistemas baseados em UNIX provavelmente necessitará de permissão `sudo`.

Instalada, podemos importar sua classe `timezone` e fica fácil de pegarmos o fuso horário que queremos:

```
from datetime import datetime
from pytz import timezone

data_e_hora_atuais = datetime.now()
fuso_horario = timezone('America/Sao_Paulo')
data_e_hora_sao_paulo = data_e_hora_atuais.astimezone(fuso_horario)
data_e_hora_sao_paulo_em_texto = data_e_hora_sao_paulo.strftime('%d/%m/%Y %H:%M')

print(data_e_hora_sao_paulo_em_texto)
```

Repare que nós colocamos o `timezone` como `America/Sao_Paulo`. Mas e se quisermos saber outras possibilidades? É possível ver a lista de fusos horários suportados pelo `pytz` iterando sobre o **`pytz.all_timezones`**:

```
import pytz

for tz in pytz.all_timezones:
    print(tz)
```

Trabalhar com datas pra gente não será mais um problema!

Mais Python e Datas

Ao longo desse post, usamos diversas vezes alguns códigos de formatação de datas, como por exemplo o padrão `%d/%m/%Y %H:%M`. Mas o que significa isso?

Esses códigos são definidos pela **documentação do strftime(3)**. Os usados em nossos exemplos são:

- **`%d`** - O dia do mês representado por um número decimal (de 01 a 31)
- **`%m`** - O mês representado por um número decimal (de 01 a 12)
- **`%Y`** - O ano representado por um número decimal incluindo o século
- **`%H`** - A hora representada por um número decimal usando um relógio de 24 horas (de 00 a 23)
- **`%M`** - O minuto representado por um número decimal (de 00 a 59)

Conclusão

Começamos o post com a necessidade de manipular datas com o Python e vimos

como fazer isso utilizando o tipo `date`. Aprendemos a juntar nossa data a um horário através da classe `datetime`, e como formatar essas informações em uma string de fácil leitura para nós.

Acabamos tendo um problema com o fuso horário e vimos como resolver com a classe `timezone` no Python 3. Também vimos uma maneira mais simples de solucionar esse problema com a biblioteca `pytz`, feita pela comunidade Python.

E aí? Vai ser mais fácil lidar com datas usando o Python a partir de agora, não acha? Gostou? Quer aprender mais? Confira a formação de **Python para web na Alura!**