اندازه‌گیری زمان اجرای کد

گاهی اوقات مهم است که بدانیم کد شما چقدر زمان می‌برد تا اجرا شود یا حداقل بدانیم آیا یک خط خاص از کد باعث کندی کل پروژه شما می‌شود یا خیر. پایتون یک ماژول زمانبندی داخلی دارد که این کار را انجام می‌دهد.

تابع یا اسکریپت مثال

در اینجا دو تابع داریم که همان کار را انجام می‌دهند، اما به روش‌های مختلفی. چطور می‌توانیم بفهمیم کدام یک کارآمدتر است؟ بیایید زمان آن را بسنجیم!

In [1]:

```
def func_one(n):
    '''
    Given a number n, returns a list of string integers
    ['0','1','2',...'n']
    '''
    return [str(num) for num in range(n)]
```

In [2]:

```
func_one(10)
```

Out[2]:

```
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

In [3]:

```
def func_two(n):
    '''
    Given a number n, returns a list of string integers
    ['0','1','2',...'n']
    '''
    return list(map(str,range(n)))
```

In [4]:

```
func_two(10)
```

Out[4]:

```
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

شروع و پایان زمان‌بندی

می‌توانیم از ماژول زمان برای محاسبه زمان گذشته اجرای کد استفاده کنیم. به خاطر دقت ماژول زمان، کد باید **حداقل** 0.1 ثانیه زمان ببرد تا کامل شود.

In [5]:

```
import time
```

In [11]:

```
# STEP 1: Get start time
start_time = time.time()
# Step 2: Run your code you want to time
result = func_one(1000000)
# Step 3: Calculate total time elapsed
end_time = time.time()

elaps = end_time - start_time
```

In [12]:

```
elaps
```

Out[12]:

```
0.23676824569702148
```

In [13]:

```
# STEP 1: Get start time
start_time = time.time()
# Step 2: Run your code you want to time
result = func_two(1000000)
# Step 3: Calculate total time elapsed
end_time = time.time()

elaps = end_time - start_time
```

In [14]:

```
elaps
```

Out[14]:

```
0.16420912742614746
```

ماژول timeit

چه اگر دو بخش کد داشته باشیم که به اندازه کافی سریع هستند، تفاوت از روش time.time() کافی نیست که بفهمیم کدامیک سریعتر است. در این حالت، می‌توانیم از ماژول timeit استفاده کنیم.

ماژول timeit دو رشته دریافت می‌کند، یک عبارت (stmt) و یک تنظیم. سپس کد تنظیم را اجرا کرده و کد stmt را برخی از n بار اجرا کرده و میانگین طول زمان آن را گزارش می‌دهد.

In [15]:

```
import timeit
```

تنظیمات (هر چیزی که قبل از آن باید تعریف شود، مانند تابع‌های def).

In [17]:

```
setup = '''
def func_one(n):
    return [str(num) for num in range(n)]
'''
```

In [18]:

```
stmt = 'func_one(100)'
```

In [19]:

```
timeit.timeit(stmt,setup,number=100000)
```

Out[19]:

```
1.6600021999329329
```

حالا بیایید تابع func_two را ۱۰،۰۰۰ بار اجرا کرده و طول زمان آن را مقایسه کنیم.

In [20]:

```
setup2 = '''
def func_two(n):
    return list(map(str,range(n)))
'''
```

In [21]:

```
stmt2 = 'func_two(100)'
```

In [22]:

```
timeit.timeit(stmt2,setup2,number=100000)
```

Out[22]:

```
1.3273243999574333
```

به نظر می‌آید که func_two کارآمدتر است. شما می‌توانید تعداد اجراهای بیشتری را مشخص کنید اگر می‌خواهید تفاوت برای توابع با عملکرد سریع‌تر را واضح کنید.

In [23]:

```
timeit.timeit(stmt,setup,number=1000000)
```

Out[23]:

```
17.106991000007838
```

In [24]:

```
timeit.timeit(stmt2,setup2,number=1000000)
```

Out[24]:

```
12.608633200172335
```

تایمینگ کد خود را با روش "جادویی" جوپیتر انجام دهید

توجه: این روش فقط در جوپیتر قابل استفاده است و دستور magic باید در بالای سلول با هیچ چیزی بالای آن (حتی کد مورد نظری که به صورت توضیحات نوشته شده است) قرار نگیرد

In [25]:

```
%%timeit
func_one(100)
```

```
17.1 µs ± 236 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

In [26]:

```
%%timeit
func_two(100)
```

```
12.9 µs ± 79.9 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops eac
h)
```

برای مطالعه بیشتر به مستندات زیر مراجعه کنید: https://docs.python.org/3/library/timeit.html (https://docs.python.org/3/library/timeit.html)