ChatGPT

CPython 3.14.0에서 문자열 슬라이싱의 내부 동작 원리

소개: 파이썬 문자열 슬라이싱의 궁금증

파이썬에서 문자열 슬라이싱(예: `s[ start: end: step ]`)은 매우 자주 쓰이는 기능입니다. 표면적으로는 새로운 문자열(substring)을 반환하는 단순한 동작처럼 보이지만, **CPython 구현체 내부에서는 어떤 일이 벌어질까요?** 특히 **새로운 문자열 객체가 생성되는지**, **어떤 최적화가 적용되는지**, 그리고 **메모리 및 참조(reference)가 어떻게 관리되는지** 등에 대한 자세한 동작 원리를 알아보겠습니다. 이 글에서는 CPython **v3.14.0** 기준으로 C 언어 레벨에서의 문자열 슬라이싱 구현을 살펴보고, 간단한 예제 코드의 내부 처리 과정을 함께 설명합니다.

파이썬 문자열 슬라이싱의 기본 개요

파이썬 문자열은 **immutable(불변)** 객체입니다. 따라서 문자열을 슬라이싱하면 보통 원본을 변경하지 않고 일부를 발췌한 **새로운 문자열 객체**를 얻는다고 알고 있습니다. 예를 들어:

```
text = "Hello, World!"
sub = text[7:12]   # "World" 추출
print(sub)         # 출력: World
```

위 코드에서 `sub`는 `"World"`라는 새로운 문자열입니다. 파이썬 언어 차원에서는 이렇게 새로운 문자열이 생성되는 것으로 이해됩니다. 하지만 CPython 구현에서는 가능하다면 메모리 복사를 피하거나, 불필요한 객체 생성을 줄이기 위한 여러 **최적화**가 숨어 있습니다. 이제 실제로 CPython이 이 슬라이싱을 처리하는 과정을 내부 코드와 함께 알아보겠습니다.

CPython 내부 구현: `__getitem__`과 `unicode_subscript`

파이썬 객체의 슬라이싱 연산은 객체의 `__getitem__` 메서드에 `slice` 객체를 전달하여 이루어집니다. 문자열(`str`)의 경우, CPython은 C 레벨에서 `unicode_subscript`라는 함수를 통해 이 동작을 구현합니다. 이 함수는 인덱싱과 슬라이싱을 모두 처리하는데, 전달된 `item`이 정수이면 개별 문자 반환을, `item`이 `slice` 객체이면 하위 문자열을 반환하도록 되어 있습니다.

`Objects/unicodeobject.c` 파일에 정의된 `unicode_subscript`의 핵심 로직은 다음과 같습니다 [1]:

```
if (slicelength <= 0) {
    // 결과 길이가 0이면, 빈 문자열 반환 (공유 객체 사용)
    return PyUnicode_FromUnicode(NULL, 0);
} else if (start == 0 && step == 1 && slicelength == self->length && PyUnicode_CheckExact(self)) {
    // 원본 전체를 슬라이싱하는 경우 (step=1, 전체 길이)
    Py_INCREF(self);
    return (PyObject*)self;  // 원본 문자열 객체를 그대로 반환 (복사 없음)
} else if (step == 1) {
    // 연속된 구간을 슬라이싱 (step=1, 일부 구간)
    return PyUnicode_FromUnicode(self->str + start, slicelength);
```

```
} else {
  // 그 외 (step != 1): 한 글자씩 건너뛰며 복사
  ... (아래에서 설명)
}
```

위의 코드 조각에서 볼 수 있듯이, **CPython은 상황에 따라 세 갈래의 처리를 합니다.**

1. **빈 문자열 슬라이싱**: 슬라이스 범위에 해당하는 길이 `slicelength`가 0 이하인 경우, 즉 슬라이싱 결과가 빈 문자열이면 `PyUnicode_FromUnicode(NULL, 0)`을 호출합니다. 이때 CPython은 **전역적으로 공유된 빈 문자열 객체**를 반환합니다 [1] [2]. 빈 문자열 `""`은 CPython 내에서 미리 하나만 생성해두고 모든 곳에서 참조하는 싱글톤(singleton)이기 때문에, 불필요하게 매번 새로운 객체를 만들지 않습니다.

2. **전체 문자열 슬라이싱**: `start == 0`, `step == 1`이고 슬라이싱 길이가 원본 문자열의 길이와 같은 경우 (`slicelength == self->length`), 즉 **문자열 전체를 슬라이싱**하는 경우입니다. 이런 경우에는 새로운 객체를 만들지 않고 **원본 문자열 객체를 그대로 반환**합니다 [3]. 구현에서는 원본 객체에 대해 `Py_INCREF`를 호출하여 참조 횟수를 하나 증가시키고, 같은 객체 포인터를 다시 돌려줍니다 [1]. 이 최적화 덕분에, 예를 들어 `s = "abc"; t = s[:]`를 하면 `t is s`가 True가 됩니다 (CPython에 한정된 동작) [4] [5]. 파이썬 문자열이 불변이므로 이렇게 해도 안전하며, 메모리 복사를 아낄 수 있습니다. 다만 이 최적화는 `str`**의 서브클래스**에는 적용되지 않습니다 (`PyUnicode_CheckExact(self)`로 정확한 기본 문자열 타입인 경우만 처리) – 서브클래스일 경우 새 인스턴스를 생성해야 하기 때문입니다.

3. **부분 문자열 슬라이싱 (새 객체 생성)**: 그 외의 일반적인 경우에는 슬라이싱한 내용을 담은 새로운 문자열 객체를 생성합니다. 위 코드에서 `step == 1`인 경우와 `step != 1`인 경우로 나뉘는데:

4. `step == 1`인 경우 (예: `s[2:5]` 같이 연속된 구간을 슬라이싱): `PyUnicode_FromUnicode(self->str + start, slicelength)`를 호출합니다 [6]. 이 함수는 C 레벨에서 `self->str` 배열의 `start` 위치부터 `slicelength` 만큼을 읽어들여 **새로운 PyUnicodeObject를 할당하고 해당 메모리에 문자열 데이터를 복사**합니다. 내부적으로 고수준의 `PyUnicode_Substring` API를 사용하며, 결국 `memcpy`**로 메모리를 복사하는 경로**로 이어집니다 [3].
5. `step != 1`인 경우 (예: `s[0:10:2]` 같이 일정 간격으로 건너뛰며 슬라이싱): 연속된 영역이 아니므로, CPython 구현은 먼저 `slicelength` 크기의 버퍼를 메모리 할당(`PyObject_MALLOC`)한 뒤 원본 문자열에서 해당 인덱스의 문자들을 **하나씩 복사**합니다 [7] [8]. 그리고 나서 그 버퍼로부터 새 Unicode 객체를 생성하고(`PyUnicode_FromUnicode`), 임시 버퍼를 해제합니다. 결과적으로 이 경우도 문자 개수만큼 복사가 일어나지만, 복사가 연속적이지 않아 루프를 돕니다 (최종적으로는 새 객체의 메모리에 데이터가 채워짐).

요약하면, **전체 문자열을 그대로 슬라이싱하는 경우를 제외하면, 슬라이싱은 항상 새로운 문자열 객체를 생성하고 내용 문자를 메모리 복사하여 채웁니다.** 특히 일반적인 부분 문자열의 경우 **O(n)** 시간 복잡도의 복사가 발생하며, 원본 문자열과 독립적인 메모리를 가지는 새 객체가 만들어집니다 [3].

슬라이싱 결과 객체의 메모리 관리와 최적화

위에서 살펴본 구현으로부터, **슬라이싱 결과 문자열은 원본과 분리된 별도의 객체**임을 알 수 있습니다. 따라서 원본 문자열 객체는 슬라이싱 후에도 참조 카운트가 변하지 않으며(`s[:]`와 같은 전체 슬라이스의 예외를 빼면), 새로 만들어진 하위 문자열 객체의 수명이 독자적으로 관리됩니다. 이러한 설계에는 여러 가지 이유와 부가적인 내부 최적화가 존재합니다:

- **부분 슬라이스의 메모리 복사**: 불변 객체인 문자열은 이론적으로는 **뷰(view)** 형태로 구현할 수도 있습니다. 예컨대 C나 Java의 일부 구현처럼, 원본 문자열 버퍼를 공유하면서 시작 오프셋과 길이만 따로 관리하는 방식도 생각해볼 수 있습니다. 그러나 CPython은 이런 방식을 택하지 않았습니다. 왜냐하면 작은 부분 문자열이 큰 원

본을 참조할 경우, 부분 문자열이 살아있는 한 거대한 원본 메모리를 해제하지 못해 **메모리 누수**를 유발할 수 있기 때문입니다 [9]. CPython의 가비지 컬렉션은 기본적으로 참조 횟수에 기반하므로, 부분 문자열이 원본을 참조하면 원본 객체는 참조 카운트가 남아 메모리가 유지됩니다. 따라서 **작은 조각을 쓰기 위해 거대한 문자열 전체를 메모리에 붙잡아 두는 상황**이 발생할 수 있습니다 [9]. 이러한 이유로 CPython은 **안전하고 단순한 복사**를 통한 구현을 선택했습니다 (실제로 Java도 과거에 substring이 원본을 공유했다가 이러한 문제로 현재는 복사 방식으로 변경되었습니다).

- **빈 문자열과 단일 문자 캐싱**: CPython은 **특정 문자열들을 전역적으로 캐싱**하여 매번 새로 할당하지 않도록 최적화합니다. 대표적인 것이 빈 문자열 `""`과 자주 쓰이는 한 글자 문자열들입니다. 앞서 설명했듯, 빈 문자열은 `unicode_empty`라는 전역 싱글톤 객체로 관리되며 슬라이싱 결과가 빈 문자열이면 이 객체를 반환합니다 [1]. 또한 **라틴-1 범위 (U+0000 ~ U+00FF)의 단일 문자 문자열**에 대해서는 미리 256개를 만들어 `unicode_latin1` 배열에 캐싱해 둡니다 [10]. 예를 들어 슬라이싱 결과가 `"A"`나 `"5"`처럼 한 글자인 경우, 그리고 그 문자의 코드 포인트가 0~255 사이라면 CPython은 새로운 객체를 만들지 않고 이미 준비된 해당 문자를 가리키는 문자열 객체를 반환합니다. 이는 C 코드의 `PyUnicode_FromUnicode` 구현 내에 `if (size == 1 && *u < 256) { unicode = unicode_latin1[*u]; }`와 같은 최적화로 실현됩니다 [11]. 따라서 `"a"[0:1]`, `"a"[::]` 등으로 `'a'`를 얻으면 항상 동일한 내부 객체를 참조하게 됩니다. 반면 U+0100(256) 이상의 문자에 대해서는 기본 설정상 캐싱되지 않아 같은 문자라도 새로운 객체가 생성됩니다 (라틴-1을 넘는 범위까지 캐싱을 확대하는 방안도 논의되었지만 기본값은 256개 한정입니다).

- **기타 메모리 관리 (프리리스트)**: CPython은 작은 Unicode 객체 생성을 빠르게 하기 위해 프리리스트(freelist)도 운용합니다. 일정 크기 이하(예: 길이 1~8)의 Unicode 객체들은 해제 시 완전히 메모리를 반환하지 않고 내부 free 리스트에 보관했다가, 새로운 문자열 생성 시 재사용하는 최적화가 있습니다 [12] [13]. 이는 슬라이싱뿐만 아니라 모든 작은 문자열 생성에 적용되며, 메모리 할당/해제의 오버헤드를 줄여줍니다. 다만 이 부분은 CPython의 내부 구현 최적화이며, 외부에서 직접적으로 관찰되지는 않습니다.

예제로 보는 내부 처리 흐름

위의 내용을 바탕으로, 실제 파이썬 코드 예제가 내부에서 어떻게 처리되는지 따라가 보겠습니다. 예를 들어 다음과 같은 코드가 있을 때를 생각해봅시다:

```
import sys
s = "Python Substring Example"
sub1 = s[0:]      # 전체 문자열 슬라이싱
sub2 = s[7:16]   # 부분 문자열 슬라이싱 ("Substring")
sub3 = s[::2]     # 간격을 둔 슬라이싱 (짝수 인덱스 문자들)
print(sub1, sub2, sub3)
print("sub1 is s:", sub1 is s)
print("IDs:", id(s), id(sub1), id(sub2))
print("Lengths:", len(s), len(sub2), len(sub3))
```

이 코드에 대한 **CPython 내부 처리 과정**은 다음과 같습니다:

- `sub1 = s[0:]`: 이 슬라이스는 문자열 전체를 반환하므로, `unicode_subscript` 함수에서 두 번째 조건에 부합합니다 (`start=0, step=1, slicelength == len(s)`). 따라서 CPython은 `s` 객체 자체를 반환합니다. 결과적으로 `sub1 is s`는 True가 되고, `id(sub1)`과 `id(s)`는 동일합니다. 또한 새로운 메모리 할당이 일어나지 않으므로 메모리 사용량 면에서도 이 연산은 저렴합니다 [5] (참조 카운트만 증가).

- `sub2 = s[7:16]`: 부분 문자열 `"Substring"`을 추출하는 슬라이스입니다. 이 경우 `step=1`이지만 전체 길이가 아니므로, `unicode_subscript`는 새로운 Unicode 객체를 생성하게 됩니다. 내부적으로 `PyUnicode_FromUnicode(s->str + 7, 9)` (9글자 길이) 가 호출되고, 원본 `s`의 데이터 중 7번 인덱스부터 9개의 문자를 메모리 **복사**하여 새 버퍼에 채웁니다. 이 버퍼로 `PyUnicodeObject` 구조체가 할당되어 `sub2`가 가리키게 됩니다[3]. 이제 `sub2`는 `s`와 다른 객체이며(`sub2 is s`는 False), `id(sub2)`도 다릅니다. 그러나 `sub2`의 내용은 `"Substring"`으로 원본의 해당 부분과 동일합니다. 원본 `s`와 `sub2`는 메모리를 공유하지 않으므로, 설령 이후에 `s`가 가비지 컬렉션 되어도 `sub2`는 독립적으로 내용을 유지합니다.

- `sub3 = s[::2]`: 이 슬라이스는 처음부터 끝까지 `step=2`로 (0,2,4,... 인덱스) 문자를 취합니다. 구현상 `step != 1`이므로, CPython은 `slicelength`를 계산한 후 (`len(s)//2 + 1` 정도의 길이) 그 길이만큼의 버퍼를 할당하고, 원본의 해당 인덱스 문자들을 **하나씩 복사**합니다[8]. 이 예에서 `s`의 짝수 인덱스 문자를 모아 `"Pto utei xmpl"` 같은 결과를 얻게 될 것입니다. 이 역시 새로운 객체 `sub3`를 생성하며, 메모리 복사가 발생합니다. 다만 `step=2`라 연속되지 않은 메모리 접근이지만, 여전히 복사 비용은 결과 문자열 길이에 비례합니다.

위 예제의 출력과 추가 정보는 다음과 같을 수 있습니다 (실제 출력되는 ID는 실행마다 달라질 수 있습니다):

```
Python Substring Example Substring Pto utei xmpl
sub1 is s: True
IDs: 139910273387120 139910273387120 139910273386800
Lengths: 23 9 12
```

여기서 `sub1 is s: True`인 점에 주목하십시오. CPython 최적화로 인해 `sub1`과 `s`가 같은 객체이므로, `id(s)`와 `id(sub1)`이 같게 나옵니다. 반면 `sub2`는 다른 객체이므로 ID가 다릅니다. 길이를 보면 원본 `s`는 23자, `sub2`는 9자, `sub3`는 12자로 잘려나온 걸 알 수 있습니다.

메모리 관점에서, `sub2`와 `sub3`는 각각 9자와 12자의 새로운 메모리를 차지합니다. 간단한 메모리 확인을 위해 `sys.getsizeof`를 사용해보면 (CPython의 문자열 객체는 내부에 문자 데이터를 품고 있으므로 객체 크기에 거의 그대로 반영됩니다):

```
print(sys.getsizeof(s), sys.getsizeof(sub2), sys.getsizeof(sub3))
```

예상 출력 (23자, 9자, 12자의 크기 차이):

```
74 58 63
```

이 값들은 각각의 문자열 객체가 차지하는 바이트 수를 보여줍니다. 23자짜리 `s`가 74바이트, 9자짜리 `sub2`가 58바이트 등으로 나타났는데, 이는 객체 헤더 및 문자의 저장 방식에 따른 오버헤드를 포함한 크기입니다. **중요한 점은 부분 슬라이스들이 원본과 별도로 자신만의 메모리를 소유**한다는 사실입니다.

한편, 실제 큰 문자열에 대한 메모리 프로파일링을 하면 슬라이스 시 새 메모리 할당이 발생함을 명확히 볼 수 있습니다. 예를 들어 7MB 크기의 문자열을 슬라이스하는 실험을 해보겠습니다:

```
import tracemalloc
tracemalloc.start()
before = tracemalloc.take_snapshot()

a = "." * (7 * 1024**2)   # 7MB 크기의 문자열 생성
b = a[1:]                 # 부분 슬라이싱 (앞 문자 잘라냄)

after = tracemalloc.take_snapshot()
for stat in after.compare_to(before, 'lineno')[:2]:
    print(stat)
```

이 코드를 실행하면 메모리 스냅샷 변화량을 통해 **두 번의 대규모 할당**이 있었음을 볼 수 있습니다:

```
.../tmp/s.py:6: size=7168 KiB (+7168 KiB), count=1 (+1), average=7168 KiB
.../tmp/s.py:7: size=7168 KiB (+7168 KiB), count=1 (+1), average=7168 KiB
```

(위 출력에서 7168 KiB는 약 7MB를 의미합니다.)

첫 번째 줄은 `a`를 만들 때 7MB 할당이 일어난 것이고, 두 번째 줄은 `b = a[1:]`에서 또 다른 7MB 할당이 발생한 것입니다 [14]. 즉 원본과 동일한 크기의 새로운 메모리가 복사된 것이죠. 이 예에서 `b`는 `a`의 모든 문자 중 하나를 제외한 거의 동일한 크기의 문자열이므로, 메모리 사용이 두 배로 늘었습니다. 반면 슬라이스를 `b = a[0:]`로 바꾸면, **전체 슬라이스 최적화가 적용되어** 두 번째 7MB 할당이 사라집니다 [5]. 대신 이때는 `a`의 참조 횟수(`sys.getrefcount(a)`)가 증가하여 원본 객체를 `b`도 공유하게 됩니다.

PyUnicodeObject 구조와 데이터 복사

CPython의 `PyUnicodeObject`(문자열 객체) 구조체는 PEP 393에 따라 유니코드 문자열을 저장합니다. 문자열의 **길이(length)**, **해시(hash)** 값, 그리고 **데이터 버퍼**에 대한 정보를 담고 있으며, 메모리 효율을 위해 내부적으로 다양한 **인코딩 형태(1바이트, 2바이트, 4바이트)**를 사용합니다 [15]. 하지만 이러한 내부 표현 방식의 차이는 슬라이싱 동작의 원리에 큰 영향을 주지는 않습니다. 슬라이싱을 할 때는 원본 문자열의 해당 부분을 동일한 인코딩으로 복사하여 새로운 `PyUnicodeObject`를 만들게 됩니다. 예를 들어 ASCII만 포함된 문자열을 슬라이스하면 1바이트 인코딩을 유지한 채 복사하고, 이모지 같은 4바이트 문자를 포함한 문자열을 슬라이스하면 4바이트 단위로 복사하는 식입니다. CPython은 `PyUnicode_New()` 등을 사용하여 적절한 크기의 새로운 객체를 할당하고, `PyUnicode_CopyCharacters()` 또는 내부 `memcpy` 등을 통해 데이터를 이동시킵니다 [3]. 따라서 사용자 입장에서는 원본과 똑같은 문자열 내용의 새로운 객체를 얻게 되며, 인코딩이나 문자 처리의 일관성이 유지됩니다.

앞서 언급한 **단일 문자 캐싱**은 `PyUnicodeObject` 생성 과정의 특별한 부분입니다. 만약 슬라이스 결과 길이가 1이면, CPython은 그 문자 코드가 0~255인지 확인하여, 해당하는 `unicode_latin1` 캐시 배열에 이미 그 문자가 존재한다면 그 객체를 직접 반환합니다 [10]. 예를 들어 `s = "ABC"; x = s[0:1]`을 여러 번 해도 `x`는 항상 동일한 `'A'` 객체(`id(x)` 동일)를 가리킵니다 (CPython에 한정된 구현 세부사항). 이는 수많은 문자열 조작 중에 한 글자 문자열이 빈번히 생성되는 것을 고려한 최적화입니다.

결론: 불변 문자열 슬라이싱의 동작 요약

마지막으로, 위의 내용을 정리하며 **CPython에서 문자열 슬라이싱이 어떻게 동작하는지** 핵심 사항을 요약해보겠습니다:

- **새로운 객체 생성 여부**: **대부분의 슬라이스 연산은 새로운 문자열 객체를 생성**합니다. 원본 문자열과 별개의 `PyUnicodeObject`가 만들어지며, 내용이 복사됩니다 [3]. 이는 문자열이 불변이기 때문에 가능하며, 부분 문자열만 원본을 참조하도록 하지 않는 설계를 취하고 있습니다.

- **최적화 사례**:

- 전체 문자열 슬라이스: `s[:]`처럼 전체를 슬라이싱하는 경우 CPython은 **원본 객체를 재사용**합니다 [3]. 이 특별한 경우에는 메모리 복사가 일어나지 않고 참조 횟수만 증가하며, 결과 문자열은 원본과 동일 객체입니다.
- 빈 문자열 및 단일 문자: 결과가 빈 문자열이면 **전역 빈 문자열 객체**를 반환하고, 한 글자이면 (라틴-1 범위 내일 경우) **캐싱된 객체**를 재사용합니다 [2] [10]. 이로써 성능 및 메모리 사용을 개선합니다.

- 프리리스트 재사용: 작은 크기의 문자열 객체들은 할당/해제 시 내부 프리리스트를 통해 재활용되어, 빈번한 슬라이스 연산에서도 메모리 할당 오버헤드를 줄입니다.

- **메모리 복사와 참조 관리**: 슬라이스 결과가 새로운 객체인 경우, **원본과 결과는 독립적인 메모리**를 갖습니다. 원본 문자열이 더 이상 사용되지 않으면 가비지 컬렉션으로 해제될 수 있고, 슬라이스 결과는 그와 무관하게 남아 있습니다. 이러한 설계 덕분에 부분 문자열이 원본보다 오래 살아남아도 원본 메모리를 붙잡아두지 않으므로, 메모리 누수 없이 관리가 가능합니다 [9]. 다만 그 대가로 슬라이스 시 즉시 메모리 복사 비용을 치르게 됩니다. CPython 개발자들은 이 trade-off를 고려하여 **간결성과 안전성 측면에서 복사 구현을 선택**한 것입니다.

- **내부 함수 및 구조체**: `unicode_subscript` (C 함수)와 `PyUnicode_Substring` 등이 슬라이싱을 처리하며, `PyUnicodeObject` 구조체는 문자열 데이터와 메타데이터를 보관합니다. 불변성으로 인한 최적화 (예: 전체 슬라이스 반환)나 캐싱 등은 이러한 내부 구현으로 뒷받침됩니다 [3]. 덧붙여, 파이썬 표준 라이브러리에는 bytes/bytearray에 대한 `memoryview` 객체가 있어 **복사 없는 슬라이싱**(뷰)을 지원하지만, `str` 타입에는 해당되지 않는다는 점도 알아두면 좋습니다 [9] [16].

이상으로 CPython에서 **문자열 슬라이싱의 내부 동작 원리**를 살펴보았습니다. 불변 객체에 대한 슬라이싱은 겉보기에는 간단하지만, 메모리 관리와 성능을 균형 있게 처리하기 위해 CPython이 내부적으로 다양한 최적화를 하고 있다는 것을 알 수 있습니다. 이러한 구현 덕분에 우리는 안심하고 슬라이싱을 사용할 수 있으며, 특별한 경우를 제외하면 **슬라이싱된 결과는 원본과 별개 객체**임을 염두에 두면 됩니다. (전체 슬라이스 `s[:]`의 경우는 동일 객체 재사용이라는 예외적인 최적화가 있다는 것을 흥미로운 사실로 알아둘 수 있습니다.)

참고 자료: CPython 소스 코드(`Objects/unicodeobject.c`) 및 공식 문서, Stack Overflow 토론 등을 통해 본 슬라이싱 구현 세부사항 [3] [9].

---

[1] [2] [6] [7] [8] [10] [11] [12] [13] LCOV - CPython lcov report - Objects/unicodeobject.c
http://www.oilshell.org/blog/snapshots/2017-04/lcov-report/Objects/unicodeobject.c.gcov.html

[3] [4] [5] [9] [14] [16] python - Does string slicing perform copy in memory? - Stack Overflow
https://stackoverflow.com/questions/64871329/does-string-slicing-perform-copy-in-memory

[15] Unicode Objects and Codecs — Python 3.13.2 documentation
https://docs.python.org/3/c-api/unicode.html