
인자 클리닉 How-To

출시 버전 3.9.18

Guido van Rossum
and the Python development team

2월 06, 2024

Contents

1	인자 클리닉의 목표	2
2	기본 개념과 사용법	2
3	첫 번째 함수 변환하기	3
4	고급 주제	9
4.1	기호 기본값	9
4.2	인자 클리닉에서 생성한 C 함수와 변수 이름 변경하기	9
4.3	PyArg_UnpackTuple을 사용하여 함수 변환하기	10
4.4	선택적 그룹	10
4.5	“레거시 변환기” 대신 실제 인자 클리닉 변환기 사용하기	11
4.6	Py_buffer	13
4.7	고급 변환기	13
4.8	매개 변수 기본값	14
4.9	NULL 기본값	14
4.10	기본값으로 지정된 표현식	14
4.11	반환 변환기 사용하기	15
4.12	기존 함수 복제하기	16
4.13	파이썬 코드 호출하기	16
4.14	“self 변환기” 사용하기	17
4.15	사용자 정의 변환기 작성하기	17
4.16	사용자 정의 반환 변환기 작성하기	18
4.17	METH_O와 METH_NOARGS	18
4.18	tp_new와 tp_init 함수	19
4.19	클리닉 출력을 변경하고 리디렉션하기	19
4.20	#ifdef 트릭	22
4.21	파이썬 파일에서 인자 클리닉 사용하기	23
	색인	24

저자 Larry Hastings

요약

인자 클리닉(Argument Clinic)은 CPython C 파일을 위한 전 처리기입니다. 그 목적은 “내장”에 대한 인자 구문 분석 코드 작성과 관련된 모든 상용구를 자동화하는 것입니다. 이 설명서는 여러분의 첫 번째 C 함수를 인자 클리닉과 함께 작동하도록 변환하는 방법을 보여준 다음, 인자 클리닉 사용에 대한 몇 가지 고급 주제를 소개합니다.

현재 인자 클리닉은 CPython에 내부 전용으로 간주합니다. CPython 외부의 파일에 대해서는 사용이 지원되지 않으며, 향후 버전에서 이전 버전과의 호환성을 보장하지 않습니다. 즉: 여러분이 CPython에 대한 외부 C 확장을 유지한다면, 여러분의 자체 코드에서 인자 클리닉을 실험하는 것은 환영합니다. 그러나 다음 버전의 CPython과 함께 제공되는 인자 클리닉 버전은 완전히 호환되지 않고 여러분의 모든 코드를 망가뜨릴 수 있습니다.

1 인자 클리닉의 목표

인자 클리닉의 주요 목표는 CPython 내부의 모든 인자 구문 분석 코드에 대한 책임을 인수하는 것입니다. 즉, 인자 클리닉에서 작동하도록 함수를 변환할 때, 해당 함수는 더는 자체 인자 구문 분석을 수행하지 않아야 합니다 - 인자 클리닉에서 생성된 코드는 여러분에게 “블랙박스”여야 합니다, CPython이 맨 위에서 호출하고, 맨 아래에서 여러분의 코드가 호출되고, PyObject *args (그리고 아마도 PyObject *kwargs)가 여러분이 필요로 하는 C 변수와 형으로 마술처럼 변환됩니다.

인자 클리닉이 기본 목표를 달성하려면, 사용하기 쉬워야 합니다. 현재, CPython의 인자 구문 분석 라이브러리로 작업하는 것은 따분한 일이며, 놀랄 정도로 많은 장소에서 중복된 정보를 유지해야 합니다. 인자 클리닉을 사용할 때, 여러분 스스로 반복할 필요가 없습니다.

분명히, 자체적으로 새로운 문제를 만들지 않으면서 자신의 문제를 해결하지 않는 한 아무도 인자 클리닉을 사용하고 싶어 하지 않을 것입니다. 따라서 인자 클리닉이 올바른 코드를 생성하는 것이 가장 중요합니다. 코드가 더 빠르면 좋겠지만, 최소한 주요 속도 회귀를 도입해서는 안 됩니다. (인자 클리닉은 결국 대폭적인 속도 향상을 가능하게 해야 합니다 - 범용 CPython 인자 구문 분석 라이브러리를 호출하는 대신 맞춤형 인자 구문 분석 코드를 생성하도록 코드 생성기를 다시 작성할 수 있습니다. 그러면 가능한 가장 빠른 인자 구문 분석이 될 것입니다!)

또한, 인자 클리닉은 인자 구문 분석에 대한 모든 접근 방식을 사용할 수 있을 만큼 유연해야 합니다. 파이썬에는 매우 이상한 구문 분석 동작을 가진 몇 가지 함수가 있습니다; 인자 클리닉의 목표는 이들 모두를 지원하는 것입니다.

마지막으로, 인자 클리닉의 원래 동기는 CPython 내장에 대한 인트로스펙션 “서명”을 제공하는 것이었습니다. 예전에는 내장을 전달하면 인트로스펙션 조회 함수에서 예외가 발생했습니다. 인자 클리닉을 사용하면, 그것은 과거의 일입니다!

인자 클리닉과 함께 일할 때, 명심해야 할 한 가지 아이디어가 있습니다: 더 많은 정보를 제공할수록, 더 나은 작업을 수행할 수 있습니다. 인자 클리닉은 현재 비교적 간단합니다. 그러나 진화함에 따라 더 정교해질 것이며, 여러분이 제공하는 모든 정보로 많은 흥미롭고 현명한 일을 할 수 있어야 합니다.

2 기본 개념과 사용법

인자 클리닉은 CPython과 함께 제공됩니다; Tools/clinic/clinic.py에서 찾을 수 있습니다. 해당 스크립트를 실행하면, C 파일을 인자로 지정합니다:

```
$ python3 Tools/clinic/clinic.py foo.c
```

인자 클리닉은 파일을 스캔하여 다음과 같은 줄을 찾습니다:

```
/*[clinic input]
```

찾으면, 다음과 같은 줄까지 모든 것을 읽습니다:

```
[clinic start generated code]*/
```

이 두 줄 사이의 모든 것은 인자 클리닉에 대한 입력입니다. 시작과 끝 주석 줄을 포함하여, 이러한 모든 줄을 총칭하여 인자 클리닉 “블록”이라고 합니다.

인자 클리닉이 이러한 블록 중 하나를 구문 분석할 때, 출력을 생성합니다. 이 출력은 C 파일의 블록 바로 뒤에 다시 쓰이고, 체크섬이 포함된 주석이 이어집니다. 인자 클리닉 블록은 이제 다음과 같습니다:

```
/*[clinic input]
... clinic input goes here ...
[clinic start generated code]*/
... clinic output goes here ...
/*[clinic end generated code: checksum=...]*/
```

같은 파일에서 인자 클리닉을 두 번 실행하면, 인자 클리닉은 이전 출력을 버리고 새로운 체크섬 줄로 새 출력을 작성합니다. 그러나, 입력이 변경되지 않았으면, 출력도 변경되지 않습니다.

인자 클리닉 블록의 출력 부분을 수정해서는 안 됩니다. 대신, 원하는 출력을 생성할 때까지 입력을 변경하십시오. (그것이 체크섬의 목적입니다 - 누군가 출력을 변경했는지 감지하는 것, 다음에 인자 클리닉이 새로운 출력을 작성할 때 이러한 편집이 손실되기 때문입니다.)

명확성을 위해, 인자 클리닉에서 사용할 용어는 다음과 같습니다:

- 주석의 첫 번째 줄(`/*[clinic input]`)은 시작 줄(*start line*)입니다.
- 초기 주석의 마지막 줄(`[clinic start generated code]*/`)은 끝줄(*end line*)입니다.
- 마지막 줄(`/*[clinic end generated code: checksum=...]*/`)은 체크섬 줄(*checksum line*)입니다.
- 시작 줄과 끝줄 사이에 있는 것이 입력(*input*)입니다.
- 끝줄과 체크섬 줄 사이에 있는 것이 출력(*output*)입니다.
- 시작 줄에서 체크섬 줄까지 모든 텍스트는 총칭하여 블록(*block*)입니다. (인자 클리닉에서 성공적으로 처리되지 않은 블록은 아직 출력이나 체크섬 줄이 없지만, 여전히 블록으로 간주합니다.)

3 첫 번째 함수 변환하기

인자 클리닉의 작동 방식을 이해하는 가장 좋은 방법은 함수를 작동하도록 변환하는 것입니다. 다음은, 인자 클리닉에서 작동하도록 함수를 변환하기 위해 따라야 할 최소한의 단계입니다. CPython에 체크인하려는 코드의 경우, 설명서의 뒷부분에서 볼 수 있는 고급 개념(“반환 변환기”와 “self 변환기”와 같은)을 사용하여 변환 작업을 더 진행해야 합니다. 하지만 이 연습에서는 배우기 쉽도록 간단하게 유지하겠습니다.

뛰어듭시다!

0. CPython trunk의 새로 갱신된 체크 아웃으로 작업하고 있는지 확인하십시오.
1. `PyArg_ParseTuple()` 이나 `PyArg_ParseTupleAndKeywords()` 를 호출하고, 아직 인자 클리닉에서 작동하도록 변환되지 않은 파이썬 내장을 찾습니다. 제 예에서는 `_pickle.Pickler.dump()` 를 사용하고 있습니다.
2. `PyArg_Parse` 함수에 대한 호출이 다음 포맷 단위 중 하나를 사용하거나:

```
O&
O!
es
es#
et
et#
```

또는 `PyArg_ParseTuple()` 에 대한 여러 호출이 있으면, 다른 함수를 선택해야 합니다. 인자 클리닉은 이러한 모든 시나리오를 지원합니다. 그러나 이것들은 고급 주제입니다 - 첫 번째 함수로 더 간단한 것을 해봅시다.

또한, 함수가 같은 인자에 대해 다른 형을 지원하는 `PyArg_ParseTuple()` 이나 `PyArg_ParseTupleAndKeywords()` 에 대한 여러 호출이 있거나, 함수가 인자를 구문 분

석하기 위해 PyArg_Parse 함수 이외의 것을 사용하면, 인자 클리닉으로 변환하는 데 적합하지 않을 수 있습니다. 인자 클리닉은 제네릭 함수나 다형성 매개 변수를 지원하지 않습니다.

3. 함수 위에 다음과 같은 상용구를 추가하여, 블록을 만듭니다:

```
/*[clinic input]
[clinic start generated code]*/
```

4. 독스트링을 잘라내어 [clinic] 줄 사이에 붙여넣고, 적절하게 인용된 C 문자열을 만드는 모든 정크를 제거합니다. 완료되면 왼쪽 여백을 기준으로 텍스트가 80자보다 넓은 줄이 없는, 텍스트만 남게 됩니다. (인자 클리닉은 독스트링 내부의 들여쓰기를 유지합니다.)

이전 독스트링에 함수 서명처럼 보이는 첫 번째 줄이 있으면, 해당 줄을 버립니다. (독스트링은 이것이 더는 필요하지 않습니다 - 향후 내장에 help() 를 사용할 때, 첫 번째 줄은 함수의 서명에 따라 자동으로 구축됩니다.)

샘플:

```
/*[clinic input]
Write a pickled representation of obj to the open file.
[clinic start generated code]*/
```

5. 독스트링에 “요약” 줄이 없으면, 인자 클리닉이 불평합니다. 그러니 하나 있도록 합시다. “요약” 줄은 독스트링의 시작 부분에 있는 단일 80열 줄로 구성된 단락이어야 합니다.

(예제 독스트링은 요약 줄로만 구성되어서, 이 단계에서 샘플 코드를 변경할 필요가 없습니다.)

6. 독스트링 위에, 함수 이름을 입력한 다음, 빈 줄을 입력합니다. 이것은 함수의 파이썬 이름이어야 하며, 함수에 대한 전체 점표기법 경로여야 합니다 - 모듈 이름으로 시작하고, 모든 하위 모듈을 포함해야 하며, 함수가 클래스의 메서드이면 클래스 이름도 포함해야 합니다.

샘플:

```
/*[clinic input]
_pickle.Pickler.dump

Write a pickled representation of obj to the open file.
[clinic start generated code]*/
```

7. 이 C 파일에서 해당 모듈이나 클래스가 인자 클리닉과 함께 처음 사용된 것이면, 모듈 및/또는 클래스를 선언해야 합니다. 적절한 인자 클리닉 위생법은 인클루드 파일과 정적 객체가 상단에 가는 것과 같은 방식으로 C 파일의 상단 근처에 있는 별도의 블록에 이를 선언하는 것을 선호합니다. (샘플 코드에서는 서로 옆에 있는 두 블록만 표시합니다.)

클래스와 모듈의 이름은 파이썬에서 보는 이름과 같아야 합니다. PyModuleDef나 PyTypeObject에 정의된 이름을 적절하게 확인하십시오.

클래스를 선언할 때, C에서 해당 형의 두 가지 측면을 지정해야 합니다: 이 클래스의 인스턴스에 대한 포인터에 사용할 형 선언, 그리고 이 클래스를 위한 PyTypeObject에 대한 포인터.

샘플:

```
/*[clinic input]
module _pickle
class _pickle.Pickler "PicklerObject *" "&Pickler_Type"
[clinic start generated code]*/

/*[clinic input]
_pickle.Pickler.dump

Write a pickled representation of obj to the open file.
[clinic start generated code]*/
```

8. 각 매개 변수를 함수에 선언합니다. 각 매개 변수는 자체 줄을 가져야 합니다. 모든 매개 변수 줄은 함수 이름과 독스트링에서 들여쓰기 되어야 합니다.

이러한 매개 변수 줄의 일반적인 형식은 다음과 같습니다:

```
name_of_parameter: converter
```

매개 변수에 기본값이 있으면, 변환기(converter) 뒤에 추가하십시오:

```
name_of_parameter: converter = default_value
```

“기본값”에 대한 인자 클리닉의 지원은 매우 정교합니다; 자세한 내용은 아래의 기본값에 관한 섹션을 참조하십시오.

매개 변수 아래에 빈 줄을 추가합니다.

“변환기(converter)”는 무엇일까요? C에서 사용되는 변수의 형과, 실행 시간에 파이썬 값을 C 값으로 변환하는 방법을 모두 설정합니다. 지금은 이전 코드를 인자 클리닉으로 더 쉽게 이식할 수 있도록 고안된 편의 문법인 “레저시 변환기”를 사용할 것입니다.

매개 변수마다, `PyArg_Parse()` format 인자에서 해당 매개 변수의 “포맷 단위”를 복사하고 그것을 (따옴표로 묶은 문자열로) 변환기로 지정하십시오. (“포맷 단위”는 인자 구문 분석 함수에 변수 형과 변환 방법을 알려주는 format 매개 변수의 1~3문자 부분 문자열에 대한 공식 이름입니다. 포맷 단위에 대한 자세한 내용은 arg-parsing을 참조하십시오.)

z#과 같은 다중 문자 포맷 단위의 경우, 전체 2~3문자 문자열 전체를 사용합니다.

샘플:

```
/*[clinic input]
module _pickle
class _pickle.Pickler "PicklerObject *" "&Pickler_Type"
[clinic start generated code]*/

/*[clinic input]
_pickle.Pickler.dump

    obj: 'O'

Write a pickled representation of obj to the open file.
[clinic start generated code]*/
```

9. 함수의 포맷 문자열에 |가 있으면 (일부 매개 변수에 기본값이 있음을 의미합니다), 무시할 수 있습니다. 인자 클리닉은 기본값이 있는지에 따라 어떤 매개 변수가 선택적인지 유추합니다.

함수의 포맷 문자열에 \$가 있으면 (키워드 전용 인자를 취함을 의미합니다), 첫 번째 키워드 전용 인자 앞에 *를 별도의 줄로 지정하고 매개 변수 줄과 같게 들여쓰기합니다.

(`_pickle.Pickler.dump`에는 둘 다 없어서, 샘플은 변경되지 않습니다.)

10. 기존 C 함수가 `PyArg_ParseTuple()` 을 호출하면 (`PyArg_ParseTupleAndKeywords()` 가 아니라), 모든 인자는 위치 전용입니다.

인자 클리닉에서 모든 매개 변수를 위치 전용으로 표시하려면, 마지막 매개 변수 뒤에 /를 추가하고 매개 변수 줄과 같게 들여쓰기합니다.

현재 이것은 전부 아니면 전무입니다; 모든 매개 변수가 위치 전용이거나, 아무것도 아닙니다. (향후 인자 클리닉에서 이 제한을 완화할 수 있습니다.)

샘플:

```
/*[clinic input]
module _pickle
class _pickle.Pickler "PicklerObject *" "&Pickler_Type"
[clinic start generated code]*/

/*[clinic input]
_pickle.Pickler.dump
```

(다음 페이지에 계속)

```

    obj: '0'
    /

Write a pickled representation of obj to the open file.
[clinic start generated code]*/

```

11. 매개 변수마다 매개 변수별 독스트링을 작성하는 것이 유용합니다. 그러나 매개 변수별 독스트링은 선택 사항입니다; 원한다면 이 단계를 건너뛸 수 있습니다.

매개 변수별 독스트링을 추가하는 방법은 다음과 같습니다. 매개 변수별 독스트링의 첫 번째 줄은 매개 변수 정의보다 더 들여 써야 합니다. 이 첫 번째 줄의 왼쪽 여백은 전체 매개 변수별 독스트링에 대한 왼쪽 여백을 설정합니다; 작성하는 모든 텍스트는 이 양만큼 내어 쓰게 됩니다. 원한다면 여러 줄에 걸쳐, 원하는 만큼 텍스트를 작성할 수 있습니다.

샘플:

```

/*[clinic input]
module _pickle
class _pickle.Pickler "PicklerObject *" "&Pickler_Type"
[clinic start generated code]*/

/*[clinic input]
_pickle.Pickler.dump

    obj: '0'
        The object to be pickled.
    /

Write a pickled representation of obj to the open file.
[clinic start generated code]*/

```

12. 파일을 저장하고 닫은 다음, 그것에 대해 Tools/clinic/clinic.py를 실행합니다. 운 좋게도 모든 것이 작동했습니다—이제 블록에 출력이 있고 .c.h 파일이 생성되었습니다! 텍스트 편집기에서 파일을 다시 열어 다음을 확인하십시오:

```

/*[clinic input]
_pickle.Pickler.dump

    obj: '0'
        The object to be pickled.
    /

Write a pickled representation of obj to the open file.
[clinic start generated code]*/

static PyObject *
_pickle_Pickler_dump(PicklerObject *self, PyObject *obj)
/*[clinic end generated code: output=87ecad1261e02ac7 input=552eb1c0f52260d9]*/

```

명백히, 인자 클리닉이 출력을 생성하지 않았다면, 입력에서 에러를 발견했기 때문입니다. 인자 클리닉이 불평 없이 파일을 처리할 때까지 에러를 수정하고 재시도하십시오.

가독성을 위해, 대부분의 글루(glue) 코드는 .c.h 파일에 생성되었습니다. 일반적으로 클리닉 모듈 블록 바로 뒤에서, 원본 .c 파일에 포함해야 할 필요가 있습니다:

```
#include "clinic/_pickle.c.h"
```

13. 인자 클리닉에서 생성한 인자 구문 분석 코드가 기본적으로 기존 코드와 같은지 다시 확인합니다.

먼저, 두 곳에서 같은 인자 구문 분석 함수를 사용하는지 확인하십시오. 기존 코드는 PyArg_ParseTuple() 이나 PyArg_ParseTupleAndKeywords() 를 호출해야 합니다; 인자 클

리닉에서 생성한 코드가 정확히 같은 함수를 호출하는지 확인합니다.

둘째, `PyArg_ParseTuple()` 이나 `PyArg_ParseTupleAndKeywords()` 에 전달된 포맷 문자열은 콜론이나 세미콜론까지 기존 함수에서 손으로 쓴 것과 정확히 같아야 합니다.

(인자 클리닉은 항상 : 뒤에 함수 이름이 있는 포맷 문자열을 생성합니다. 기존 코드의 포맷 문자열이 ;로 끝나면 (사용법 도움말을 제공하기 위해), 이 변경 사항은 무해합니다 - 걱정하지 마십시오.)

셋째, 포맷 단위가 두 개의 인자(가령 길이 변수, 인코딩 문자열 또는 변환 함수에 대한 포인터)를 요구하는 매개 변수의 경우, 두 번째 인자가 두 호출 간에 정확히 같은지 확인하십시오.

넷째, 블록의 출력 부분 내부에 이 내장에 적합한 정적 `PyMethodDef` 구조체를 정의하는 전 처리기 매크로가 있습니다:

```
#define __PICKLE_PICKLER_DUMP_METHODDEF \
{"dump", (PyCFunction)__pickle_Pickler_dump, METH_O, __pickle_Pickler_dump__
↪doc__},
```

이 정적 구조체는 이 내장의 기존 정적 `PyMethodDef` 구조체와 정확히 같아야 합니다.

이러한 항목 중 어떤 식으로건 다른 항목이 있으면, 인자 클리닉 함수 명세를 조정하고 같아질 때까지 `Tools/clinic/clinic.py`를 다시 실행합니다.

14. 출력의 마지막 줄은 “impl” 함수의 선언임에 유의하십시오. 여기가 내장 구현이 들어가는 곳입니다. 수정 중인 함수의 기존 프로토타입을 삭제하십시오, 하지만 여는 중괄호는 그대로 둡니다. 이제 인자 구문 분석 코드와 인자를 덤프하는 모든 변수의 선언을 삭제합니다. 이제 어떤 식으로 파이썬 인자가 이 impl 함수에 대한 인자가 되는지 주목하십시오; 구현에서 이러한 변수에 다른 이름을 사용했다면, 수정하십시오.

좀 괴상하니, 반복합니다. 이제 코드는 다음과 같아야 합니다:

```
static return_type
your_function_impl(...)
/*[clinic end generated code: checksum=...]*/
{
...
}
```

인자 클리닉은 체크섬 줄과 그 바로 위에 함수 프로토타입을 생성했습니다. 함수와 내부 구현에 대한 여는 (그리고 닫는) 중괄호를 작성해야 합니다.

샘플:

```
/*[clinic input]
module _pickle
class _pickle.Pickler "PicklerObject *" "&Pickler_Type"
[clinic start generated code]*/
/*[clinic end generated code:↵
↪checksum=da39a3ee5e6b4b0d3255bfef95601890afd80709]*/

/*[clinic input]
_pickle.Pickler.dump

    obj: 'O'
        The object to be pickled.
    /

Write a pickled representation of obj to the open file.
[clinic start generated code]*/

PyDoc_STRVAR(__pickle_Pickler_dump__doc__,
"Write a pickled representation of obj to the open file.\n"
"\n"
...
static PyObject *
```

(다음 페이지에 계속)


```

_pickle_Pickler_dump_impl(PicklerObject *self, PyObject *obj)
/*[clinic end generated code:~
↪checksum=3bd30745bf206a48f8b576a1da3d90f55a0a4187]*/
{
    /* Check whether the Pickler was initialized correctly (issue3664).
       Developers often forget to call __init__() in their subclasses, which
       would trigger a segfault without this check. */
    if (self->write == NULL) {
        PyErr_Format(PicklingError,
                     "Pickler.__init__() was not called by %s.__init__()",
                     Py_TYPE(self)->tp_name);
        return NULL;
    }

    if (_Pickler_ClearBuffer(self) < 0)
        return NULL;

    ...

```

15. 이 함수에 대한 PyMethodDef 구조체의 매크로를 기억하십니까? 이 함수에 대한 기존 PyMethodDef 구조체를 찾아 매크로에 대한 참조로 바꿉니다. (내장이 모듈 스코프에 있으면, 아마도 파일의 끝부분에 가까울 것입니다; 내장이 클래스 메서드이면, 아마도 아래에 있지만, 상대적으로 구현에 가까울 것입니다.)

매크로 본문에는 후행 쉼표가 포함되어 있음에 유의하십시오. 따라서 기존의 정적 PyMethodDef 구조체를 매크로로 바꿀 때, 끝에 쉼표를 추가하지 마십시오.

샘플:

```

static struct PyMethodDef Pickler_methods[] = {
    __PICKLE_PICKLER_DUMP_METHODDEF
    __PICKLE_PICKLER_CLEAR_MEMO_METHODDEF
    {NULL, NULL} /* sentinel */
};

```

16. 컴파일한 다음, 회귀 테스트 스위트의 관련 부분을 실행합니다. 이 변경으로 인해 새로운 컴파일 시간 경고나 에러가 발생해서는 안 되며, 파이썬의 동작에 대해 외부에서 볼 수 있는 변경 사항이 없어야 합니다.

한 가지 차이점이 있습니다: 함수에 대해 실행되는 `inspect.signature()` 는 이제 유효한 서명을 제공해야 합니다!

축하합니다, 인자 클리닉과 함께 작동하는 첫 번째 함수를 이식했습니다!

4 고급 주제

이제 인자 클리닉으로 작업한 경험이 있고, 몇 가지 고급 주제를 살펴볼 시간입니다.

4.1 기호 기본값

매개 변수에 제공하는 기본값은 임의의 표현식이 될 수 없습니다. 현재 다음이 명시적으로 지원됩니다:

- 숫자 상수 (정수와 부동 소수점)
- 문자열 상수
- True, False 및 None
- `sys.maxsize`와 같은 간단한 기호 상수, 모듈 이름으로 시작해야 합니다

궁금한 점이 있을 때를 위해, 이것은 `Lib/inspect.py`의 `from_builtin()`에서 구현됩니다.

(미래에는, `CONSTANT - 1`과 같은 완전한 표현식을 허용하기 위해, 더 정교해질 필요가 있습니다.)

4.2 인자 클리닉에서 생성한 C 함수와 변수 이름 변경하기

인자 클리닉은 자동으로 생성되는 함수의 이름을 지정합니다. 생성된 이름이 기존 C 함수의 이름과 충돌하면, 때때로 이로 인해 문제가 발생할 수 있습니다. 쉬운 해결책이 있습니다: C 함수에 사용되는 이름을 재정의하는 것입니다. 함수 선언 줄에 키워드 "as"를 추가한 다음 사용하려는 함수 이름을 추가하면 됩니다. 인자 클리닉은 기본 (생성된) 함수에 해당 함수 이름을 사용한 다음, 끝에 "_impl"을 추가하고 이를 impl 함수의 이름에 사용합니다.

예를 들어, `pickle.Pickler.dump`에 대해 생성된 C 함수 이름을 바꾸려면, 다음과 같이 됩니다:

```
/*[clinic input]
pickle.Pickler.dump as pickler_dumper
...
```

이제 기본 함수의 이름은 `pickler_dumper()` 이고 impl 함수의 이름은 `pickler_dumper_impl()` 이 됩니다.

마찬가지로, 매개 변수에 특정 파이썬 이름을 지정하려고 하지만, 해당 이름이 C에서 불편할 수 있는 경우 문제가 있을 수 있습니다. 인자 클리닉에서는 같은 "as" 문법을 사용하여, 파이썬과 C에서 매개 변수에 다른 이름을 지정할 수 있도록 합니다:

```
/*[clinic input]
pickle.Pickler.dump

    obj: object
    file as file_obj: object
    protocol: object = NULL
    *
    fix_imports: bool = True
```

여기서, 파이썬에서 사용되는 이름(서명과 keywords 배열에서)은 `file`이지만, C 변수의 이름은 `file_obj`입니다.

이것을 사용하여 `self` 매개 변수의 이름도 바꿀 수 있습니다!

4.3 PyArg_UnpackTuple을 사용하여 함수 변환하기

PyArg_UnpackTuple()로 인자를 구문 분석하는 함수를 변환하려면, 각 인자를 object로 지정하여 모든 인자를 작성하면 됩니다. type 인자를 지정하여 형을 적절하게 캐스트 할 수 있습니다. 모든 인자는 위치 전용으로 표시되어야 합니다(마지막 인자 뒤에 /를 자체 줄로 추가하십시오).

현재 생성된 코드는 PyArg_ParseTuple()을 사용하지만, 곧 변경됩니다.

4.4 선택적 그룹

일부 레거시 함수는 인자를 구문 분석하는 데 까다로운 접근 방식을 사용합니다: 위치 인자의 수를 계산한 다음 switch 문을 사용하여 위치 인자의 수에 따라 여러 PyArg_ParseTuple() 호출 중 하나를 호출합니다. (이러한 함수는 키워드 전용 인자를 받아들일 수 없습니다.) 이 접근 방식은 PyArg_ParseTupleAndKeywords()가 만들어지기 전에 선택적 인자를 시뮬레이션하는 데 사용되었습니다.

이 접근 방식을 사용하는 함수는 종종 PyArg_ParseTupleAndKeywords(), 선택적 인자 및 기본 값을 사용하도록 변환될 수 있지만, 항상 가능한 것은 아닙니다. 이러한 레거시 함수 중 일부에는 PyArg_ParseTupleAndKeywords()가 직접 지원하지 않는 동작이 있습니다. 가장 명백한 예는 필수 인자의 좌 측에 선택적 인자가 있는 내장 함수 range()입니다! 또 다른 예는 항상 함께 지정되어야 하는 두 개의 인자 그룹이 있는 curses.window.addch()입니다. (인자는 x와 y라고 합니다; 함수를 호출할 때 x를 전달하면 y도 전달해야 합니다 - 그리고 x를 전달하지 않으면 y도 전달할 수 없습니다.)

어쨌든, 인자 클리닉의 목표는 의미를 변경하지 않고 기존의 모든 CPython 내장에 대한 인자 구문 분석을 지원하는 것입니다. 따라서 인자 클리닉은 선택적 그룹(*optional groups*)이라는 것을 사용하여, 구문 분석에 대한 이러한 대체 접근 방식을 지원합니다. 선택적 그룹은 모두 함께 전달되어야 하는 인자 그룹입니다. 필수 인자의 왼쪽 또는 오른쪽에 있을 수 있습니다. 위치 전용 매개 변수에만 사용할 수 있습니다.

참고: 선택적 그룹은 오직 PyArg_ParseTuple()을 여러 번 호출하는 함수를 변환할 때 사용하려는 것입니다! 인자를 구문 분석하기 위해 다른 접근 방식을 사용하는 함수는 거의 절대 선택적 그룹을 사용하여 인자 클리닉으로 변환되지 않습니다. 선택적 그룹을 사용하는 함수는 현재 파이썬에서 정확한 서명을 가질 수 없습니다, 파이썬이 개념을 이해하지 못하기 때문입니다. 가능한 한 선택적 그룹을 사용하지 마십시오.

선택적 그룹을 지정하려면, 함께 그룹화하려는 매개 변수 앞에 [를 단독 줄로 추가하고, 이러한 매개 변수 뒤에 단독 줄로]를 추가합니다. 예를 들어, curses.window.addch가 선택적 그룹을 사용하여 처음 두 매개 변수와 마지막 매개 변수를 선택적으로 만드는 방법은 다음과 같습니다:

```
/*[clinic input]
curses.window.addch

    [
        x: int
            X-coordinate.
        y: int
            Y-coordinate.
    ]

    ch: object
        Character to add.

    [
        attr: long
            Attributes for the character.
    ]
/

...
```

노트:

- 모든 선택적 그룹에 대해, 하나의 추가 매개 변수가 `impl` 함수로 전달되어 그룹을 나타냅니다. 매개 변수는 `group_{direction}_{number}`라는 이름의 정수입니다. 여기서 `{direction}`은 그룹이 필수 매개 변수 앞인지 뒤인지에 따라 `right`나 `left`이고, `{number}`는 그룹이 필수 매개 변수에서 얼마나 멀리 떨어져 있는지를 나타내는 단조 증가 하는 숫자(1에서 시작)입니다. `impl`이 호출될 때, 이 그룹이 사용되지 않았으면 이 매개 변수는 0으로 설정되고, 이 그룹이 사용되면 0이 아닌 값으로 설정됩니다. (사용했다는 표현은, 매개 변수가 이 호출에서 인자를 받았는지를 의미합니다.)
- 필수 인자가 없으면, 선택적 그룹은 필수 인자의 오른쪽에 있는 것처럼 작동합니다.
- 모호한 경우, 인자 구문 분석 코드는 왼쪽(필수 매개 변수 앞)의 매개 변수를 선호합니다.
- 선택적 그룹은 위치 전용 매개 변수 만 포함할 수 있습니다.
- 선택적 그룹은 오직 레거시 코드를 위한 것입니다. 새 코드에 선택적 그룹을 사용하지 마십시오.

4.5 “레거시 변환기” 대신 실제 인자 클리닉 변환기 사용하기

시간을 절약하고, 인자 클리닉으로의 첫 번째 이식을 달성하는 데 필요한 학습량을 최소화하기 위해, 위의 연습에서는 “레거시 변환기”를 사용하도록 지시합니다. “레거시 변환기”는 기존 코드를 인자 클리닉으로 더 쉽게 이식 할 수 있도록 명시적으로 설계된 편의 기능입니다. 명확하게 말하면, 파이썬 3.4 용 코드를 이식할 때는 사용할 수 있습니다.

그러나, 장기적으로 우리는 모든 블록이 변환기를 위한 인자 클리닉의 실제 문법을 사용하기를 원할 것입니다. 왜일까요? 몇 가지 이유가 있습니다:

- 적절한 변환기는 읽기가 훨씬 쉽고 의도가 명확합니다.
- 인자가 필요한데, 레거시 변환기 문법이 인자 지정을 지원하지 않아서, “레거시 변환기”로 지원되지 않는 일부 포맷 단위가 있습니다.
- 미래에 우리는 `PyArg_ParseTuple()` 이 지원하는 것에 제한되지 않는 새로운 인자 구문 분석 라이브러리를 가질 수 있습니다; 이러한 유연성은 레거시 변환기를 사용하는 매개 변수에는 제공되지 않을 것입니다.

따라서, 약간의 추가 노력을 꺼리지 않는다면, 레거시 변환기 대신 일반 변환기를 사용하십시오.

간단히 말해서, 인자 클리닉 (비 레거시) 변환기의 문법은 파이썬 함수 호출처럼 보입니다; 그러나, 함수에 대한 명시적 인자가 없으면 (모든 함수가 기본값을 취함), 괄호를 생략할 수 있습니다. 따라서 `bool` 과 `bool()` 은 정확히 같은 변환기입니다.

인자 클리닉 변환기에 대한 모든 인자는 키워드 전용입니다. 모든 인자 클리닉 변환기는 다음 인자를 받아들입니다:

c_default C에서 정의될 때 이 매개 변수의 기본값. 특히, 이것은 “구문 분석 함수”에서 선언된 변수의 초기화자가 됩니다. 이것을 사용하는 방법은 [기본값에 관한 섹션](#)을 참조하십시오. 문자열로 지정됩니다.

annotation 이 매개 변수의 어노테이션 값. **PEP 8**은 파이썬 라이브러리가 어노테이션을 사용하지 않도록 요구하므로, 현재 지원되지 않습니다.

또한, 일부 변환기는 추가 인자를 받아들입니다. 다음은 의미와 함께, 이러한 인자들의 목록입니다:

accept 파이썬 형 (그리고 의사 형도 가능)의 집합; 이는 허용 가능한 파이썬 인자를 이러한 형의 값으로 제한합니다. (이것은 범용 기능이 아닙니다; 일반적으로 레거시 변환기 표에 표시된 특정 형 리스트만 지원합니다.)

`None`을 받아들여려면, 이 집합에 `NoneType`을 추가하십시오.

bitwise 부호 없는 정수에 대해서만 지원됩니다. 이 파이썬 인자의 네이티브 정숫값은 음수 값에 대해서조차 범위 검사 없이 매개 변수에 기록됩니다.

converter object 변환기에서만 지원됩니다. 이 객체를 네이티브 형으로 변환하는 데 사용할 C “변환기 함수”의 이름을 지정합니다.

encoding 문자열에 대해서만 지원됩니다. 이 문자열을 파이썬 `str` (유니코드) 값에서 C `char *` 값으로 변환할 때 사용할 인코딩을 지정합니다.

subclass_of object 변환기에 대해서만 지원됩니다. 파이썬 값은 C로 표현된 파이썬 형의 서브 클래스여야 합니다.

type object와 self 변환기에 대해서만 지원됩니다. 변수를 선언하는 데 사용할 C형을 지정합니다. 기본 값은 "PyObject *"입니다.

zeroes 문자열에 대해서만 지원됩니다. 참이면, 값 내에 내장된 NUL 바이트('\0')가 허용됩니다. 문자열의 길이는 문자열 매개 변수 바로 뒤에 <parameter_name>_length 라는 이름의 매개 변수로 impl 함수에 전달됩니다.

가능한 모든 인자 조합이 작동하는 것은 아님에 유의하십시오. 일반적으로 이러한 인자는 특정 동작을 갖는 특정 PyArg_ParseTuple 포맷 단위에 의해 구현됩니다. 예를 들어, 현재 bitwise=True를 지정하지 않고 unsigned_short를 호출할 수 없습니다. 이것이 작동하리라 생각하는 것이 합리적이지만, 이러한 의미는 기존 포맷 단위에 매핑되지 않습니다. 그래서 인자 클리닉은 이것을 지원하지 않습니다. (또는, 적어도 아직은 아닙니다.)

다음은 레저시 변환기를 실제 인자 클리닉 변환기에 매핑하는 표입니다. 왼쪽에는 레저시 변환기가 있고, 오른쪽에는 교체할 텍스트가 있습니다.

'B'	unsigned_char (bitwise=True)
'b'	unsigned_char
'c'	char
'C'	int (accept={str})
'd'	double
'D'	Py_complex
'es'	str (encoding='name_of_encoding')
'es#'	str (encoding='name_of_encoding', zeroes=True)
'et'	str (encoding='name_of_encoding', accept={bytes, bytearray, str})
'et#'	str (encoding='name_of_encoding', accept={bytes, bytearray, str}, zeroes=True)
'f'	float
'h'	short
'H'	unsigned_short (bitwise=True)
'i'	int
'I'	unsigned_int (bitwise=True)
'k'	unsigned_long (bitwise=True)
'K'	unsigned_long_long (bitwise=True)
'l'	long
'L'	long long
'n'	Py_ssize_t
'O'	object
'O!'	object (subclass_of='&PySomething_Type')
'O&'	object (converter='name_of_c_function')
'p'	bool
'S'	PyBytesObject
's'	str
's#'	str (zeroes=True)
's*'	Py_buffer (accept={buffer, str})
'U'	unicode
'u'	Py_UNICODE
'u#'	Py_UNICODE (zeroes=True)
'w*'	Py_buffer (accept={rwbuffer})
'Y'	PyByteArrayObject
'y'	str (accept={bytes})
'y#'	str (accept={robuffer}, zeroes=True)
'y*'	Py_buffer
'Z'	Py_UNICODE (accept={str, NoneType})
'Z#'	Py_UNICODE (accept={str, NoneType}, zeroes=True)

다음 페이지에 계속

표 1 - 이전 페이지에서 계속

'z'	str(accept={str, NoneType})
'z#'	str(accept={str, NoneType}, zeroes=True)
'z*'	Py_buffer(accept={buffer, str, NoneType})

예를 들어, 적절한 변환기를 사용하는 샘플 `pickle.Pickler.dump`는 다음과 같습니다:

```
/*[clinic input]
pickle.Pickler.dump

    obj: object
        The object to be pickled.
    /

Write a pickled representation of obj to the open file.
[clinic start generated code]*/
```

실제 변환기의 한 가지 장점은 레거시 변환기보다 유연하다는 것입니다. 예를 들어, `unsigned_int` 변환기(그리고 모든 `unsigned_` 변환기)는 `bitwise=True` 없이 지정될 수 있습니다. 기본 동작은 값에 대해 범위 검사를 수행하며, 음수를 허용하지 않습니다. 레거시 변환기로는 그렇게 할 수 없습니다!

인자 클리닉은 사용 가능한 모든 변환기를 보여줍니다. 각 변환기에 대해 허용되는 모든 매개 변수와 각 매개 변수의 기본값이 표시됩니다. 전체 목록을 보려면 `Tools/clinic/clinic.py --converters`를 실행하십시오.

4.6 Py_buffer

`Py_buffer` 변환기(또는 `'s*'`, `'w*'`, `'*y'` 또는 `'z*'` 레거시 변환기)를 사용할 때, 제공된 버퍼에서 `PyBuffer_Release()`를 호출하지 않아야 합니다. 인자 클리닉은(구문 분석 함수에서) 이를 수행하는 코드를 생성합니다.

4.7 고급 변환기

고급이기 때문에 처음에는 건너뛴 포맷 단위를 기억하십니까? 다음은 이것도 처리하는 방법입니다.

트릭은, 모든 포맷 단위가 인자를 취한다는 것입니다- 변환 함수, 형 또는 인코딩을 지정하는 문자열. (그러나 “레거시 변환기”는 인자를 지원하지 않습니다. 이것이 바로 첫 번째 함수에서 건너뛴 이유입니다.) 포맷 단위에 지정한 인자는 이제 변환기에 대한 인자입니다; 이 인자는 `converter` (O&의 경우), `subclass_of` (O!의 경우) 또는 `encoding` (e로 시작하는 모든 포맷 단위의 경우)입니다.

`subclass_of`를 사용할 때, `object()`에 대한 다른 사용자 정의 인자를 사용하고 싶을 수도 있습니다: 매개 변수에 실제로 사용되는 형을 설정할 수 있는 `type`. 예를 들어, 객체가 `PyUnicode_Type`의 서브 클래스 인지 확인하려면, `object(type='PyUnicodeObject *', subclass_of='&PyUnicode_Type')` 변환기를 사용할 수 있습니다.

인자 클리닉을 사용할 때 발생할 수 있는 한 가지 문제: e로 시작하는 포맷 단위에 대해 일부 가능한 유연성을 제거합니다. `PyArg_Parse` 호출을 직접 작성할 때, 이론적으로 실행 시간에 `PyArg_ParseTuple()`에 전달할 인코딩 문자열을 결정할 수 있습니다. 그러나 이제 이 문자열은 인자 클리닉 처리 시점에 하드 코딩되어야 합니다. 이 제한은 의도적입니다; 이 포맷 단위를 지원하는 것을 훨씬 쉽게 만들고, 향후 최적화를 허용할 수 있습니다. 이 제한은 비합리적으로 보이지 않습니다; CPython 자체는 항상 포맷 단위가 e로 시작하는 매개 변수에 대해 정적 하드 코딩된 인코딩 문자열을 전달합니다.

4.8 매개 변수 기본값

매개 변수의 기본값은 여러 값 중 하나일 수 있습니다. 가장 간단하게는, 문자열, 정수 또는 부동 소수점 리터럴일 수 있습니다:

```
foo: str = "abc"
bar: int = 123
bat: float = 45.6
```

또한 파이썬의 내장 상수를 사용할 수 있습니다:

```
yep: bool = True
nope: bool = False
nada: object = None
```

또한 다음 섹션에 설명된 NULL과 단순 표현식 기본값에 대한 특별 지원도 있습니다.

4.9 NULL 기본값

문자열과 객체 매개 변수의 경우, None으로 설정하여 기본값이 없음을 나타낼 수 있습니다. 그러나, 이는 C 변수가 Py_None으로 초기화됨을 의미합니다. 편의상, 이 이유로 NULL이라는 특수 값이 있습니다: 파이썬의 관점에서 보면 None의 기본값처럼 동작하지만, C 변수는 NULL로 초기화됩니다.

4.10 기본값으로 지정된 표현식

매개 변수의 기본값은 단순한 리터럴 값 이상이 될 수 있습니다. 수학 연산자를 사용하고 객체의 어트리뷰트를 조회하는 전체 표현식이 될 수 있습니다. 그러나, 이 지원은 일부 명확하지 않은 의미로 인해 간단하지 않습니다.

다음 예를 고려하십시오:

```
foo: Py_ssize_t = sys.maxsize - 1
```

sys.maxsize는 플랫폼마다 다른 값을 가질 수 있습니다. 따라서 인자 클리닉은 단순히 해당 표현식을 로컬에서 평가하고 C로 하드 코딩할 수 없습니다. 따라서 사용자가 함수의 서명을 요청할 때, 실행 시간에 평가되는 방식으로 기본값을 저장합니다.

식을 평가할 때 사용할 수 있는 이름 공간은 무엇입니까? 내장이 온 모듈의 컨텍스트에서 평가됩니다. 따라서, 모듈에 “max_widgets”라는 어트리뷰트가 있으면, 간단히 사용할 수 있습니다:

```
foo: Py_ssize_t = max_widgets
```

심볼이 현재 모듈에서 발견되지 않으면, sys.modules를 찾는 것으로 폴백 됩니다. 이것이 예를 들어 sys.maxsize를 찾는 방법입니다. (사용자가 인터프리터에 로드할 모듈을 미리 알지 못하므로, 파이썬 자체에 의해 미리 로드된 모듈로 제한하는 것이 가장 좋습니다.)

실행 시간에만 기본값을 평가한다는 것은 인자 클리닉이 올바른 동등한 C 기본값을 계산할 수 없음을 의미합니다. 그래서 여러분은 그것을 명시적으로 말할 필요가 있습니다. 표현식을 사용할 때, 변환기에 대한 c_default 매개 변수를 사용하여 C에서 동등한 표현식도 지정해야 합니다:

```
foo: Py_ssize_t(c_default="PY_SSIZE_T_MAX - 1") = sys.maxsize - 1
```

또 다른 복잡함: 인자 클리닉은 여러분이 제공한 표현식이 유효한지를 미리 알 수 없습니다. 올바르게 보이는지 확인하기 위해 구문 분석하지만, 실제로 올바른지 알 수는 없습니다. 실행 시간에 유효하다고 보장되는 값을 지정하기 위해 표현식을 사용할 때 매우 주의해야 합니다!

마지막으로, 표현식은 정적 C값으로 표현할 수 있어야 하므로, 유효한 표현식에는 많은 제한이 있습니다. 다음은 사용이 허용되지 않는 파이썬 기능 목록입니다:

- 함수 호출.

- 인라인 if 문 (3 if foo else 5).
- 자동 시퀀스 언패킹 (*[1, 2, 3]).
- 리스트/집합/딕셔너리 컴프리헨션과 제너레이터 표현식.
- 튜플/리스트/집합/딕셔너리 리터럴.

4.11 반환 변환기 사용하기

기본적으로 인자 클리닉이 생성하는 `impl` 함수는 `PyObject *`를 반환합니다. 그러나 여러분의 C 함수는 종종 어떤 C형을 계산한 다음, 마지막 순간에 `PyObject *`로 변환합니다. 인자 클리닉은 파이썬 형의 입력을 네이티브 C형으로 변환하는 작업을 처리합니다 - 반환 값을 네이티브 C형에서 파이썬 형으로 변환하지 않을 이유가 무엇입니까?

이것이 “반환 변환기(return converter)”가 하는 일입니다. C형을 반환하도록 `impl` 함수를 변경한 다음, 생성된 (`impl`이 아닌) 함수에 코드를 추가하여 해당 값을 적절한 `PyObject *`로 변환합니다.

반환 변환기의 문법은 매개 변수 변환기의 것과 유사합니다. 함수 자체에 대한 반환 어노테이션처럼 반환 변환기를 지정합니다. 반환 변환기는 매개 변수 변환기와 거의 같게 작동합니다; 인자를 취하고, 인자는 모두 키워드 전용이며, 기본 인자를 변경하지 않으면 괄호를 생략할 수 있습니다.

(함수에 대해 "as"와 반환 변환기를 모두 사용하면, "as"가 반환 변환기 앞에 와야 합니다.)

반환 변환기를 사용할 때 한 가지 추가적인 문제가 있습니다: 에러가 발생했음을 어떻게 표시합니까? 일반적으로, 함수는 성공에 대해 유효한 (NULL이 아닌) 포인터를 반환하고, 실패에 대해 NULL을 반환합니다. 그러나 정수 반환 변환기를 사용하면, 모든 정수가 유효합니다. 인자 클리닉은 어떻게 에러를 감지할까요? 해결책: 각 반환 변환기는 에러를 나타내는 특수 값을 묵시적으로 찾습니다. 해당 값을 반환하고 에러가 설정되면 (`PyErr_Occurred()`는 참값을 반환합니다), 생성된 코드가 에러를 전파합니다. 그렇지 않으면 정상일 때처럼 반환되는 값을 인코딩합니다.

현재 인자 클리닉은 단지 몇 가지 반환 변환기만 지원합니다:

```
bool
int
unsigned int
long
unsigned int
size_t
Py_ssize_t
float
double
DecodeFSDefault
```

이들 중 어느 것도 매개 변수를 취하지 않습니다. 처음 세 개의 경우, -1을 반환하여 에러를 나타냅니다. `DecodeFSDefault`의 경우, 반환형은 `const char *`입니다; 에러를 나타내기 위해 NULL 포인터를 반환합니다.

(`Py_None`에 대한 참조 횟수를 늘리지 않고, 성공 시 `Py_None`을 반환하거나 실패 시 NULL을 반환할 수 있는, 실험적인 `NoneType` 변환기도 있습니다. 사용할 가치가 있을 만큼 명확성을 추가할 수 있을지 모르겠습니다.)

인자 클리닉이 지원하는 모든 반환 변환기를 매개 변수(있다면)와 함께 보려면, 전체 목록을 위해 `Tools/clinic/clinic.py --converters`를 실행하십시오.

4.12 기존 함수 복제하기

유사해 보이는 함수가 여러 개이면, 클리닉의 “복제(clone)” 기능을 사용할 수 있습니다. 기존 함수를 복제할 때, 다음을 재사용합니다:

- 다음을 포함하는 매개 변수
 - 그들의 이름,
 - 모든 매개 변수와 함께, 그들의 변환기,
 - 그들의 기본값,
 - 그들의 매개 변수별 독스트링,
 - 그들의 종류(*kind*) (위치 전용, 위치-키워드 또는 키워드 전용인지), 그리고
- 반환 변환기.

원래 함수에서 복사되지 않는 유일한 것은 독스트링입니다; 문법은 새 독스트링을 지정할 수 있도록 합니다.

다음은 함수 복제 문법입니다:

```
/*[clinic input]
module.class.new_function [as c_basename] = module.class.existing_function

Docstring for new_function goes here.
[clinic start generated code]*/
```

(함수는 다른 모듈이나 클래스에 있을 수 있습니다. 두 함수에 전체 경로를 사용해야 함을 예시하기 위해 샘플에 `module.class`를 작성했습니다.)

죄송합니다, 함수를 부분적으로 복제하거나, 함수를 복제한 다음 수정하는 문법은 없습니다. 복제는 전부 아니면 전무입니다.

또한, 복제하려는 함수는 현재 파일에 이전에 정의되어 있어야 합니다.

4.13 파이썬 코드 호출하기

나머지 고급 주제에서는 C 파일에서 파이썬 코드를 작성하고 인자 클리닉의 실행 시간 상태를 수정해야 합니다. 이것은 간단합니다: 파이썬 블록을 정의하기만 하면 됩니다.

파이썬 블록은 인자 클리닉 함수 블록과 다른 구분자 줄을 사용합니다. 다음과 같이 보입니다:

```
/*[python input]
# python code goes here
[python start generated code]*/
```

파이썬 블록 내부의 모든 코드는 구문 분석될 때 실행됩니다. 블록 내부에서 `stdout`에 기록된 모든 텍스트는 블록 뒤의 “출력”으로 리디렉션됩니다.

예를 들어, 다음은 C 코드에 정적 정수 변수를 추가하는 파이썬 블록입니다:

```
/*[python input]
print('static int __ignored_unused_variable__ = 0;')
[python start generated code]*/
static int __ignored_unused_variable__ = 0;
/*[python checksum:...]*/
```

4.14 “self 변환기” 사용하기

인자 클리닉은 기본 변환기를 사용하여 “self” 매개 변수를 자동으로 추가합니다. 이 매개 변수의 type을 형을 선언할 때 지정한 “인스턴스에 대한 포인터”로 자동 설정합니다. 그러나, 인자 클리닉의 변환기를 재정의하고 직접 지정할 수 있습니다. 자신의 self 매개 변수를 블록의 첫 번째 매개 변수로 추가하고, 변환기가 self_converter나 서브 클래스의 인스턴스가 되도록 하십시오.

요점은 무엇일까요? 이렇게 하면 self 형을 재정의하거나, 다른 기본 이름을 지정할 수 있습니다.

self를 캐스트 하려는 사용자 정의 형을 어떻게 지정할까요? self에 대해 같은 형의 함수가 하나나 두 개만 있으면, 인자 클리닉의 기존 self 변환기를 직접 사용하여, 사용할 형을 type 매개 변수로 전달할 수 있습니다:

```
/*[clinic input]

_pickle.Pickler.dump

    self: self(type="PicklerObject *")
    obj: object
/

Write a pickled representation of the given object to the open file.
[clinic start generated code]*/
```

반면에, self에 대해 같은 형을 사용할 함수가 많으면, self_converter를 서브 클래스화하지만 type 멤버를 재정의하는 자체 변환기를 만드는 것이 가장 좋습니다:

```
/*[python input]
class PicklerObject_converter(self_converter):
    type = "PicklerObject *"
[python start generated code]*/

/*[clinic input]

_pickle.Pickler.dump

    self: PicklerObject
    obj: object
/

Write a pickled representation of the given object to the open file.
[clinic start generated code]*/
```

4.15 사용자 정의 변환기 작성하기

이전 섹션에서 암시했듯이... 자신만의 변환기를 작성할 수 있습니다! 변환기는 CConverter를 상속하는 단순한 파이썬 클래스입니다. 사용자 정의 변환기의 주요 목적은 O& 포맷 단위를 사용하는 매개 변수가 있을 때입니다- 이 매개 변수를 구문 분석한다는 것은 PyArg_ParseTuple() “변환기 함수”를 호출하는 것을 의미합니다.

변환기 클래스의 이름은 *something*_converter여야 합니다. 이름이 이 규칙을 따르면, 변환기 클래스가 인자 클리닉에 자동으로 등록됩니다: 그 이름은 _converter 접미사가 제거된 클래스 이름이 됩니다. (이는 메타 클래스로 수행됩니다.)

CConverter.__init__를 서브 클래스 해서는 안 됩니다. 대신, converter_init() 함수를 작성해야 합니다. converter_init()는 항상 self 매개 변수를 받아들입니다; 그 후에, 모든 추가 매개 변수는 반드시 키워드 전용이어야 합니다. 인자 클리닉의 변환기에 전달된 모든 인자는 여러분의 converter_init()로 전달됩니다.

서브 클래스에 지정하고 싶은 CConverter의 추가 멤버가 있습니다. 현재 목록은 다음과 같습니다:

type 이 변수에 사용할 C형. type은 형을 지정하는 파이썬 문자열이어야 합니다, 예를 들어 int. 포인터형이면, 형 문자열은 ' *'로 끝나야 합니다.

default 이 매개 변수의 파이썬 기본값 (파이썬 값). 또는 기본값이 없으면 매직 값 unspecified.

py_default 파이썬 코드에 나타날 default (문자열). 또는 기본값이 없으면 None.

c_default C 코드에 나타날 default (문자열). 또는 기본값이 없으면 None.

c_ignored_default 기본값은 없지만, 기본값을 지정하지 않으면 “초기화되지 않은 변수” 경고를 발생시킬 수 있을 때, C 변수를 초기화하는 데 사용되는 기본값. 이는 옵션 그룹을 사용할 때 쉽게 발생할 수 있습니다 - 적절하게 작성된 코드는 실제로 이 값을 사용하지 않지만, 변수는 impl에 전달되며, C 컴파일러는 초기화되지 않은 값의 “사용”에 대해 불평합니다. 이 값은 항상 비어 있지 않은 문자열이어야 합니다.

converter C 변환기 함수의 이름 (문자열).

impl_by_reference 불리언 값. 참이면, 인자 클리닉은 변수를 impl 함수에 전달할 때 변수 이름 앞에 &를 추가합니다.

parse_by_reference 불리언 값. 참이면, 인자 클리닉은 변수를 PyArg_ParseTuple()에 전달할 때 변수 이름 앞에 &를 추가합니다.

다음은 Modules/zlibmodule.c에서 온, 사용자 정의 변환기의 가장 간단한 예입니다:

```
/*[python input]

class ssize_t_converter(CConverter):
    type = 'Py_ssize_t'
    converter = 'ssize_t_converter'

[python start generated code]*/
/*[python end generated code: output=da39a3ee5e6b4b0d input=35521e4e733823c7]*/
```

This block adds a converter to Argument Clinic named `ssize_t`. Parameters declared as `ssize_t` will be declared as type `Py_ssize_t`, and will be parsed by the 'O&' format unit, which will call the `ssize_t_converter` converter function. `ssize_t` variables automatically support default values.

더욱 정교한 사용자 정의 변환기는 사용자 정의 C 코드를 삽입하여 초기화와 정리를 처리할 수 있습니다. CPython 소스 트리에서 사용자 정의 변환기의 더 많은 예제를 볼 수 있습니다; 문자열 CConverter에 대해 C 파일을 grep 하십시오.

4.16 사용자 정의 반환 변환기 작성하기

사용자 정의 반환 변환기를 작성하는 것은 사용자 정의 변환기를 작성하는 것과 매우 유사합니다. 반환 변환기 자체가 훨씬 간단하기 때문에 다소 간단하다는 점만 다릅니다.

반환 변환기는 CReturnConverter를 서브 클래스 해야 합니다. 아직 널리 사용되지 않기 때문에, 사용자 정의 반환 변환기의 예는 아직 없습니다. 자체 반환 변환기를 작성하려면, Tools/clinic/clinic.py, 특히 CReturnConverter와 모든 서브 클래스의 구현을 읽으십시오.

4.17 METH_O와 METH_NOARGS

METH_O를 사용하는 함수를 변환하려면, 함수의 단일 인자가 object 변환기를 사용하고 있는지 확인하고, 인자를 위치 전용으로 표시하십시오:

```
/*[clinic input]
meth_o_sample

    argument: object
    /
[clinic start generated code]*/
```

METH_NOARGS를 사용하는 함수를 변환하려면, 인자를 지정하지 마십시오.

여전히 self 변환기, 반환 변환기를 사용하고, METH_O를 위한 객체 변환기에 type 인자를 지정할 수 있습니다.

4.18 tp_new와 tp_init 함수

tp_new와 tp_init 함수를 변환할 수 있습니다. 적절하게 __new__나 __init__로 이름을 지정하십시오. 참고:

- __new__에 대해 생성된 함수 이름은 기본적으로 그런 것처럼 __new__로 끝나지 않습니다. 유효한 C 식별자로 변환된 클래스의 이름일 뿐입니다.
- 이러한 함수에 대해 PyMethodDef #define이 생성되지 않습니다.
- __init__ 함수는 PyObject *가 아니라 int를 반환합니다.
- 독스트링을 클래스 독스트링으로 사용합니다.
- __new__와 __init__ 함수는 항상 args와 kwargs 객체를 모두 받아들여야 하지만, 변환할 때 이러한 함수에 대해 원하는 서명을 지정할 수 있습니다. (함수가 키워드를 지원하지 않으면, 생성된 구문 분석 함수에서 받게 되면 예외가 발생합니다.)

4.19 클리닉 출력을 변경하고 리디렉션하기

기존의 수작업으로 편집한 C 코드에 클리닉의 출력을 산재시키는 것은 불편할 수 있습니다. 운 좋게도, 클리닉은 구성 가능합니다: 나중에 (또는 이전에!) 인쇄하기 위해 출력을 버퍼링하거나, 별도의 파일에 출력을 쓸 수 있습니다. 클리닉의 생성된 출력의 모든 줄에 접두사나 접미사를 추가할 수도 있습니다.

이러한 방식으로 클리닉의 출력을 변경하면 가독성에 도움이 될 수 있지만, 형이 정의되기 전에 형을 사용하는 클리닉 코드가 발생하거나, 정의되기 전에 클리닉에서 생성된 코드를 사용하려고 시도할 수 있습니다. 이러한 문제는 파일에서 선언을 재정렬하거나, 클리닉에서 생성된 코드가 있는 곳으로 이동하여 쉽게 해결할 수 있습니다. (이것이 클리닉의 기본 동작이 모든 것을 현재 블록으로 출력하는 이유입니다; 많은 사람이 이것이 가독성을 방해한다고 생각하지만, 사용 전 정의 문제를 고치기 위해 코드를 재배열할 필요가 없습니다.)

몇 가지 용어를 정의하는 것으로 시작하겠습니다:

field 이 문맥에서, 필드는 클리닉 출력의 하위 섹션입니다. 예를 들어, PyMethodDef 구조체의 #define은 methoddef_define이라는 필드입니다. 클리닉에는 함수 정의당 출력할 수 있는 7가지 필드가 있습니다:

```
docstring_prototype
docstring_definition
methoddef_define
impl_prototype
parser_prototype
parser_definition
impl_definition
```

모든 이름은 "<a>_" 형식입니다. 여기서 "<a>"는 표현된 의미 객체(구문 분석 함수, impl 함수, 독스트링 또는 methoddef 구조체)이고 ""는 필드가 어떤 종류의 문장인지를 나타냅니다. "_prototype"으로 끝나는 필드 이름은 무언가의 실제 본문/데이터 없이 무언가의 전방 선언을 나타냅니다; "_definition"으로 끝나는 필드 이름은 무언가의 본문/데이터와 함께 무언가의 실제 정의를 나타냅니다. ("methoddef"는 특별합니다. 전 처리기 #define임을 나타내는 "_define"으로 끝나는 유일한 것입니다.)

destination 목적지(destination)는 클리닉이 출력을 쓸 수 있는 장소입니다. 5개의 내장 목적지가 있습니다:

block 기본 목적지: 현재 클리닉 블록의 출력 섹션에 인쇄됩니다.

buffer 나중에 위해 텍스트를 저장할 수 있는 텍스트 버퍼. 여기로 전송된 텍스트는 기존 텍스트의 끝에 추가됩니다. 클리닉이 파일 처리를 완료할 때 버퍼에 텍스트가 남아 있으면 에러입니다.

file 클리닉이 자동으로 만들 별도의 “클리닉 파일”입니다. 파일에 대해 선택한 파일명은 {basename}.clinic{extension}입니다. 여기서 basename과 extension에는 현재 파일에 대해 실행되는 os.path.splitext()의 출력이 대입되었습니다. (예: _pickle.c의 file 목적지는 _pickle.clinic.c에 기록됩니다.)

중요: file 목적지를 사용할 때, 생성된 파일을 반드시 체크인하는 것이 중요합니다!

two-pass buffer와 같은 버퍼. 그러나, 2 패스 버퍼는 한 번만 덤프 할 수 있으며, 모든 처리 중에 전송된 모든 텍스트를 인쇄합니다, 클리닉에서 덤프 지점 이후의 클리닉 블록에서 온 것마저도.

suppress 텍스트가 표시되지 않고 버려집니다.

클리닉은 출력을 재구성 할 수 있는 5개의 새로운 지시문을 정의합니다.

첫 번째 새 지시문은 dump입니다:

```
dump <destination>
```

이것은 명명된 목적지의 현재 내용을 현재 블록의 출력으로 덤프하고, 목적지를 비웁니다. 이것은 buffer와 two-pass 목적지에서만 작동합니다.

두 번째 새 지시문은 output입니다. output의 가장 기본적인 형태는 다음과 같습니다:

```
output <field> <destination>
```

이것은 클리닉에 *field*를 *destination*으로 출력하도록 지시합니다. output은 everything이라는 특수 메타 목적지를 지원합니다. 이 메타 목적지는 클리닉에 모든 필드를 해당 목적지로 출력하도록 지시합니다.

output에는 여러 가지 다른 함수가 있습니다:

```
output push
output pop
output preset <preset>
```

output push와 output pop을 사용하면 내부 구성 스택에 구성을 푸시하고 팝할 수 있어서, 출력 구성을 일시적으로 수정한 다음, 이전 구성을 쉽게 복원 할 수 있습니다. 변경하기 전에 푸시해서 현재 구성을 저장한 다음, 이전 구성을 복원하기 원할 때 팝 합니다.

output preset은 클리닉의 출력을 다음과 같은 여러 내장 사전 설정 구성 중 하나로 설정합니다:

block 클리닉의 원래 시작 구성. 입력 블록 바로 뒤에 모든 것을 씁니다.

parser_prototype과 docstring_prototype을 억제하고, 나머지는 모두 block에 씁니다.

file 가능한 모든 것을 “클리닉 파일”에 기록하도록 설계되었습니다. 그러면 여러분은 파일 상단 근처에서 이 파일을 #include 합니다. 이것이 작동하려면 파일을 다시 재배치해야 할 수 있습니다, 일반적으로 이것은 단지 다양한 typedef와 PyObject 정의에 대한 전방 선언을 만드는 것을 의미하지만.

parser_prototype과 docstring_prototype을 억제하고, impl_definition을 block에 쓰고 나머지는 모두 file에 씁니다.

기본 파일명은 "{dirname}/clinic/{basename}.h"입니다.

buffer 클리닉의 출력 대부분을 저장하여, 마지막에 파일에 기록합니다. 모듈이나 내장형을 구현하는 파이썬 파일의 경우, 모듈이나 내장형의 정적 구조 바로 위에 버퍼를 덤프하는 것이 좋습니다; 이것들은 일반적으로 거의 끝부분에 있습니다. 파일 중간에 정의된 정적 PyMethodDef 배열이 있으면, buffer를 사용하면 file보다 더 많은 편집이 필요할 수 있습니다.

parser_prototype, impl_prototype 및 docstring_prototype을 억제하고, impl_definition을 block에 쓰고, 나머지는 모두 file에 씁니다.

two-pass buffer 사전 설정과 유사하지만, 전방 선언을 two-pass 버퍼에 쓰고, 정의를 buffer에 씁니다. 이것은 buffer 사전 설정과 유사하지만, buffer보다 편집이 덜 필요

요할 수 있습니다. 파일 상단 근처에 two-pass 버퍼를 덤프하고, buffer 사전 설정을 사용할 때처럼 끝 근처에 buffer를 덤프하십시오.

impl_prototype 을 억제 하고, impl_definition 을 block 에 쓰고, docstring_prototype, methoddef_define 및 parser_prototype 을 two-pass에 쓰고, 나머지는 모두 buffer에 씁니다.

partial-buffer buffer 사전 설정과 유사하지만, block에 더 많은 것을 쓰고, 생성된 코드의 정말 큰 덩어리만 buffer에 씁니다. 이것은 블록의 출력에 약간 더 많은 것을 갖는 적은 비용으로, buffer의 사용 전 정의 문제를 완전히 피합니다. buffer 사전 설정을 사용할 때처럼, 끝부분에 buffer를 덤프하십시오.

impl_prototype을 억제하고, docstring_definition과 parser_definition을 buffer에 쓰고, 나머지는 모두 block에 씁니다.

세 번째 새 지시문은 destination입니다:

```
destination <name> <command> [...]
```

name이라는 목적지에서 작업을 수행합니다.

두 개의 정의된 부속 명령이 있습니다: new와 clear.

new 부속 명령은 다음과 같이 작동합니다:

```
destination <name> new <type>
```

이렇게 하면 이름이 <name>이고 형이 <type>인 새 목적지가 만들어집니다.

다음과 같은 5가지 목적지 형이 있습니다:

suppress 텍스트를 버립니다.

block 현재 블록에 텍스트를 씁니다. 이것이 클리닉이 원래 한 일입니다.

buffer 위의 “buffer” 내장 목적지와 같은, 간단한 텍스트 버퍼.

file 텍스트 파일. 파일 목적지는 다음과 같이 파일명을 빌드하는 데 사용할 템플릿인 추가 인자를 취합니다:

```
destination <name> new <type> <file_template>
```

템플릿은 내부적으로 파일명의 일부로 대체되는 세 개의 문자열을 사용할 수 있습니다:

{path} 디렉터리와 전체 파일명을 포함하는, 파일의 전체 경로.

{dirname} 파일이 있는 디렉터리의 이름.

{basename} 디렉터리를 제외한, 파일의 이름.

{basename_root} 확장자가 잘린 basename (마지막 ‘.’을 포함하지 않는 모든 것).

{basename_extension} 마지막 ‘.’ 그리고 그 이후의 모든 것. basename에 마침표가 포함되어 있지 않으면, 빈 문자열이 됩니다.

파일명에 마침표가 없으면, {basename}과 {filename}은 같고, {extension}은 비어 있습니다.”{basename}{extension}”은 항상 “{filename}.”과 정확히 같습니다.

two-pass 위의 “two-pass” 내장 목적지와 같은, 2 패스 버퍼.

clear 부속 명령은 다음과 같이 작동합니다:

```
destination <name> clear
```

목적지에서 이 지점까지 누적된 모든 텍스트를 제거합니다. (이것이 무엇에 필요한지 모르겠지만, 누군가가 실험하는 동안 유용하리라 생각했습니다.)

네 번째 새 지시문은 set입니다:

```
set line_prefix "string"
set line_suffix "string"
```

set을 사용하면 클리닉에서 두 개의 내부 변수를 설정할 수 있습니다. line_prefix는 클리닉 출력의 모든 줄 앞에 추가되는 문자열입니다; line_suffix는 클리닉 출력의 모든 줄에 뒤에 추가되는 문자열입니다.

둘 다 두 가지 포맷 문자열을 지원합니다:

{block comment start} C 파일의 시작 주석 텍스트 시퀀스인, /* 문자열로 바꿉니다.

{block comment end} C 파일의 종료 주석 텍스트 시퀀스인 */ 문자열로 바꿉니다.

마지막 새 지시문은 preserve라고 하는 여러분이 직접 사용할 필요가 없는 것입니다:

```
preserve
```

이것은 출력의 현재 내용이 수정되지 않고 유지되어야 함을 클리닉에 알려줍니다. 이는 file 파일로 출력을 덤프할 때 클리닉에서 내부적으로 사용됩니다; 클리닉 블록에서 래핑하면 클리닉이 기존 체크섬 기능을 사용하여 파일을 덮어쓰기 전에 수동으로 수정하지 않았는지 확인할 수 있습니다.

4.20 #ifdef 트릭

모든 플랫폼에서 사용할 수 없는 함수를 변환한다면, 좀 더 쉽게 만드는 데 사용할 수 있는 트릭이 있습니다. 기존 코드는 아마도 이럴 겁니다:

```
#ifdef HAVE_FUNCTIONNAME
static module_functionname(...)
{
...
}
#endif /* HAVE_FUNCTIONNAME */
```

그런 다음 하단의 PyMethodDef 구조체에서 기존 코드는 다음과 같습니다:

```
#ifdef HAVE_FUNCTIONNAME
{'functionname', ... },
#endif /* HAVE_FUNCTIONNAME */
```

이 시나리오에서는, 다음과 같이 #ifdef 안에 impl 함수의 본문을 묶어야 합니다:

```
#ifdef HAVE_FUNCTIONNAME
/*[clinic input]
module.functionname
...
[clinic start generated code]*/
static module_functionname(...)
{
...
}
#endif /* HAVE_FUNCTIONNAME */
```

그런 다음, PyMethodDef 구조체에서 앞의 세 줄을 제거하고 인자 클리닉이 생성한 매크로로 바꿉니다:

```
MODULE_FUNCTIONNAME_METHODDEF
```

(생성된 코드 내에서 이 매크로의 실제 이름을 찾을 수 있습니다. 또는 직접 계산할 수 있습니다: 블록의 첫 번째 줄에 정의된 함수 이름이지만, 마침표는 밑줄로 변경되고, 대문자로 변경되고, "_METHODDEF"를 끝에 추가합니다.)

아마도 여러분은 궁금할 겁니다: HAVE_FUNCTIONNAME 이 정의되지 않으면? MODULE_FUNCTIONNAME_METHODDEF 매크로도 정의되지 않습니다!

여기가 인자 클리닉이 매우 영리해지는 곳입니다. 실제로 인자 클리닉 블록이 `#ifdef`에 의해 비활성화될 수 있음을 감지합니다. 이 경우, 다음과 같은 약간의 추가 코드를 생성합니다:

```
#ifndef MODULE_FUNCTIONNAME_METHODDEF
#define MODULE_FUNCTIONNAME_METHODDEF
#endif /* !defined(MODULE_FUNCTIONNAME_METHODDEF) */
```

이는 매크로가 항상 작동함을 의미합니다. 함수가 정의되면, 후행 쉼표를 포함하여 올바른 구조로 바꿉니다. 함수가 정의되어 있지 않으면, 아무것도 아니게 됩니다.

그러나 이것은 한 가지 귀찮은 문제를 일으킵니다: 인자 클리닉은 “block” 출력 사전 설정을 사용할 때 이 추가 코드를 어디에 넣어야 할까요? `#ifdef`에 의해 비활성화될 수 있기 때문에, 출력 블록에 들어갈 수 없습니다. (그게 요점입니다!)

이 상황에서, 인자 클리닉은 “버퍼” 목적지에 추가 코드를 작성합니다. 이는 인자 클리닉이 불평함을 의미할 수 있습니다:

```
Warning in file "Modules/posixmodule.c" on line 12357:
Destination buffer 'buffer' not empty at end of file, emptying.
```

이 경우, 파일을 열고, 인자 클리닉이 파일에 추가한 `dump buffer` 블록(맨 아래에 있습니다)을 찾은 다음, 해당 매크로가 사용되는 `PyMethodDef` 구조체 위로 옮깁니다.

4.21 파이썬 파일에서 인자 클리닉 사용하기

인자 클리닉을 사용하여 파이썬 파일을 전처리하는 것이 실제로 가능합니다. 물론 인자 클리닉 블록을 사용하는 것은 의미가 없습니다. 출력이 파이썬 인터프리터에게 의미가 없기 때문입니다. 하지만 인자 클리닉을 사용하여 파이썬 블록을 실행하면 파이썬을 파이썬 전처리로 사용할 수 있습니다!

파이썬 주석은 C 주석과 다르기 때문에, 파이썬 파일에 포함된 인자 클리닉 블록은 약간 다르게 보입니다. 이런 식입니다:

```
#!/*[python input]
#print("def foo(): pass")
#[python start generated code]*/
def foo(): pass
#!/*[python checksum:...]*/
```

색인

Y

파이썬 향상 제안

PEP 8, 11