A. 고급 NumPy

부록 A에서는 배열 계산을 위한 NumPy 라이브러리를 좀 더 자세히 살펴보도록 하자. ndarray 자료형의 내부 구조를 상세히 알아보고 고급 배열 조작 기법과 알고리즘을 살펴본다.

A.1 ndarray 객체 구조

NumPy의 ndarray는 연속적이든 아니든 단일 형태의 데이터 블록을 다차원 배열 객체 형태로 해석할 수 있는 수단을 제공한다. dtype 이라고하는 자료형은 데이터가 실수, 정수, 불리언 혹은 다른 형인지 알려주는 역활을 한다.

ndarray가 유연한 까닭은 모든 배열 객체가 띄엄띄엄 떨어진 데이터 블록에 대한 뷰이기 때문이다. 예를 들어 arr[::2, ::-1] 배열은 어째서 데이터 복사가 일어나지 않는 것인지 의아할 수 있다. 그 이유는 ndarray는 단순한 메모리 덩어리와 dtype만을 가지는 것이 아니기 때문이다. ndarray에는 다양한 너비로 메모리 사이를 건너뛸 수 있는 보폭 정보를 포함하고 있다.

좀 더 설명하자면, ndarray는 내부적으로 다음과 같이 구성되어 있으며 다음 그림에 간략한 ndarray의 내부 구조도를 나타냈다.

- **데이터 포인터 : RAM이나 메모리 앱 파일에서 데이터의 블록**
- **dtype은 배열 내에서 값을 담는 고정된 크기를 나타낸다.**
- **배열의 모양(shape)을 알려주는 튜플**
- **하나의 차원에 따라 다음 원소로 몇 바이트 이동해야 하는지를 나타내는 stride를 담고 있는 튜플**

예를 들어 10x5 크기의 배열 모양은 (10,5) 로 표현된다.

```
In [118]: np.ones((10,5)).shape
Out[118]: (10, 5)
```

C언어 형식의 3x4x5 크기의 float(8바이트) 배열은 (160,40,8)의 stride 값을 가진다.

stride 정보를 알고 있으면 편리한데, 일반적으로 stride 값이 클수록 해당 축을 따라 연산을 수행하는 비용이 많이 들기 때문이다.

```
In [119]: np.ones((3,4,5), dtype = np.float64).strides
Out[119]: (160, 40, 8)
```

일반적인 NumPy 사용자들은 배열의 strude 값에 흥미를 가지는 경우가 드물지만, stride 값은 복사가 이뤄지짖 않는 배열의 뷰를 생성하는 데 중요한 역활을 한다. stride 값은 음수일 수도 있는데 이는 메모리상에서 뒤로 이동해야 한다는 의미다. 배열을 obj[::-1] 또는 obj[:,::-1] 형태로 잘라내는 경우가 그렇다.

A.1.1 NumPy dtype 구조

종종 배열에 담긴 값이 정수, 실수, 문자열 혹은 파이썬 객체인지 확인하는 코드를 작성할 경우가 있다. 실수에도 다양한 형태 (float 32부터 float 128까지) 가 있고, 리스트를 따라 dtype을 확인하는 과정은 꽤나 번거롭기 때문이다. 다행히도 dtype은 np.issubdtype 함수와 결합하여 사용할 수 있는 np.integer 나 np.floating 같은 부모 클래스를 가진다.

```
In [120]: ints = np.ones(10,dtype = np.uint16)

In [121]: floats = np.ones(10,dtype = np.float32)

In [122]: np.issubdtype(ints.dtype, np.integer)
Out[122]: True

In [123]: np.issubdtype(floats.dtype, np.floating)
Out[123]: True
```

특정 dtype의 모든 부모 클래스는 mro 메서드를 이용해서 확인할 수 있다.

```
In [124]: np.float64.mro()
Out[124]:
[numpy.float64,
 numpy.floating,
 numpy.inexact,
 numpy.number,
 numpy.generic,
 float,
 object]
```

B.2 운영체제와 함께 사용하기

IPython의 또 다른 중요한 기능은 운영체제 셸과 강력하게 통합되어 있다는 것이다. 즉, IPython을 종료하지 않고도 윈도우나 유닉스 (리눅스, macOS) 셸에서 일반적인 명령행에서 할 수 있는 작업이 가능하다는 뜻이다. 여기에는 셸 명령어를 실행하거나, 디렉터리를 옮기거나, 명령어의 결과를 파이썬 객체 (리스트나 문자열)에 저장하는 기능이 포함된다. 또한 간단한 셸 명령어 엘리어싱과 디렉터리 북마크 기능도 제공한다.

B.2.1 셸 명령어와 별칭

IPython에서 !로 시작하는 줄은 느낌표 다음에 있는 내용을 시스템 셸에서 실행하라는 의미다. 이 말은 rm 이나 del 명령어를 사용해서 파일을 지우거나, 디렉터리를 옮기거나, 다른 프로세스를 실행할 수 있다는 것이다.

셸 명령어의 콘솔 출력은 !를 이용해서 변수에 저장할 수 있다. 예를 들어 인터넷에 연결되어 있는 자신의 리눅스에서 IP주소를 얻어서 파이썬 변수에 대입할 수도 있다.

B.3 소프트웨어 개발 도구

지금까지 살펴본 데이터 조회와 계산에 편리한 대화형 환경에 덧붙여서 IPython은 SW 개발 환경으로도 손색 없다. 데이터 분석 어플에서는 **올바른** 코드를 작성하는 일이 가장 중요하다. 다행히도 IPython은 향상된 파이썬 pdb 디버거를 내장하고 있다. 또한 코드 실행이 빨라야 하는데 IPython은 쉽게 사용할 수 있는 코드 타이밍과 프로파일링 도구를 포함하고 있다.

B.3.1 대화형 디버거

IPython의 디버거는 pdb에 탭 자동완성 기능, 문법 갖오, 예외 트레이스백에서 각 줄에 해당하는 컨텍스트 등이 개선되었다. 코드를 디버깅하기 가장 최적인 시점은 에러가 발생한 직후다. 예외가 발생한 후 %debug 명령어를 사용하면 사후처리 디버가가 실행되고 예외가 발생한 시점의 스택 프레임 정보를 보여준다.

```
In [134]: run examples/ipython_bug.py
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
~\Desktop\data\파라데\pydata-book-2nd-edition\examples\ipython_bug.py in <module>
     13     throws_an_exception()
     14
---> 15 calling_things()

~\Desktop\data\파라데\pydata-book-2nd-edition\examples\ipython_bug.py in
calling_things()
     11 def calling_things():
     12     works_fine()
---> 13     throws_an_exception()
     14
     15 calling_things()

~\Desktop\data\파라데\pydata-book-2nd-edition\examples\ipython_bug.py in
throws_an_exception()
      7     a = 5
      8     b = 6
----> 9     assert(a + b == 10)
     10
     11 def calling_things():

AssertionError:

In [135]: %debug
> c:\users\김대현\desktop\data\파라데\pydata-book-2nd-
edition\examples\ipython_bug.py(9)throws_an_exception()
      7     a = 5
      8     b = 6
----> 9     assert(a + b == 10)
     10
     11 def calling_things():
```

디버거 안에서는 아무 파이썬 코드나 실행해볼 수 있고 각각의 스택 프레임에서 인터프리터 안에서 유지되고 있는 모든 객체와 데이터를 살펴볼 수 있다. 디폴트로 에러가 발생한 가장 아래 레벨에서부터 시작한다. u(up)과 d(down)를 눌러 스택 트레이스 사이를 이동할 수 있다.

```
ipdb> u
> c:\users\김대현\desktop\data\파라데\pydata-book-2nd-
edition\examples\ipython_bug.py(13)calling_things()
     11 def calling_things():
     12     works_fine()
---> 13     throws_an_exception()
     14
     15 calling_things()

ipdb> u
> c:\users\김대현\desktop\data\파라데\pydata-book-2nd-
edition\examples\ipython_bug.py(15)<module>()
     11 def calling_things():
     12     works_fine()
     13     throws_an_exception()
     14
---> 15 calling_things()
```

%pdb 명령어를 실행하면 예외가 발생했을 때 IPython이 자동적으로 디버거를 실행하는데 이 모드는 많은 사용자가 아주 요용하다고 생각할 것이다.

디버거는 개발하는 중에 스크립트나 함수를 싱행하는 과정에서 각 단계를 하나씩 검증하거나 브레이크포인트를 설정하고 싶을 때 쉽게 사용할 수 있다. 디버거를 실행하는 몇 가지 방법이 있는데 %run 명령에 -d 옵션을 주면 스크립트를 실행하기 전에 디버거를 먼저 실행했다. 그리고 바로 s(step)을 누르면 스크립트로 진입한다.

```
In [136]: run -d examples/ipython_bug.py
Breakpoint 1 at c:\users\김대현\desktop\data\파라데\pydata-book-2nd-
edition\examples\ipython_bug.py:1
NOTE: Enter 'c' at the ipdb>  prompt to continue execution.
> c:\users\김대현\desktop\data\파라데\pydata-book-2nd-
edition\examples\ipython_bug.py(1)<module>()
1---> 1 def works_fine():
      2     a = 5
      3     b = 6
      4     assert(a + b == 11)
      5


ipdb> s
> c:\users\김대현\desktop\data\파라데\pydata-book-2nd-
edition\examples\ipython_bug.py(6)<module>()
      4     assert(a + b == 11)
      5
----> 6 def throws_an_exception():
      7     a = 5
      8     b = 6
```

이제 부터 스크립트 파일을 어떤 식으로 동작하게 할지는 독자의 몫이다. 예를 들어 위 예제에서 works_fine 메서드를 호출하기 바로 직전에 브레이크포인트를 걸고 c(continue)를 눌러 브레이크포인트에서 멈출 때까지 스크립트를 실행할 수 있다.

```
ipdb> b 12
Breakpoint 2 at c:\users\김대현\desktop\data\파라데\pydata-book-2nd-
edition\examples\ipython_bug.py:12
ipdb> c
> c:\users\김대현\desktop\data\파라데\pydata-book-2nd-
edition\examples\ipython_bug.py(12)calling_things()
     10
     11 def calling_things():
2--> 12     works_fine()
     13     throws_an_exception()
     14
```

이제 s를 눌러 works_fine() 안으로 진입하거나, n(next) 을 눌러 works_fine()을 실행하고 다음 줄로 진행할 수 있다.

```
ipdb> n
> c:\users\김대현\desktop\data\파라데\pydata-book-2nd-
edition\examples\ipython_bug.py(13)calling_things()
     11 def calling_things():
2    12     works_fine()
---> 13     throws_an_exception()
     14
     15 calling_things()
```

이제 throws_an_exception 안으로 진입하고 에러가 발생한 다음 줄로 진행한 후 해당 범위 내에 있는 변수를 살펴보자. 디버거 명령어는 변수 이름보다 우선되므로 디버거 명령과 같은 이름의 변수가 있다면 ! 를 변수 이름 앞에 붙여서 내용을 확인할 수 있다.

```
ipdb> s
--Call--
> c:\users\김대현\desktop\data\파라데\pydata-book-2nd-
edition\examples\ipython_bug.py(6)throws_an_exception()
      4     assert(a + b == 11)
      5
----> 6 def throws_an_exception():
      7     a = 5
      8     b = 6

ipdb> n
> c:\users\김대현\desktop\data\파라데\pydata-book-2nd-
edition\examples\ipython_bug.py(7)throws_an_exception()
      5
      6 def throws_an_exception():
----> 7     a = 5
      8     b = 6
      9     assert(a + b == 10)

ipdb> n
> c:\users\김대현\desktop\data\파라데\pydata-book-2nd-
edition\examples\ipython_bug.py(8)throws_an_exception()
      6 def throws_an_exception():
      7     a = 5
----> 8     b = 6
      9     assert(a + b == 10)
     10

ipdb> n
> c:\users\김대현\desktop\data\파라데\pydata-book-2nd-
edition\examples\ipython_bug.py(9)throws_an_exception()
      7     a = 5
      8     b = 6
----> 9     assert(a + b == 10)
     10
     11 def calling_things():

ipdb> !a
5
ipdb> !b
6
```

대화형 디버거는 많은 연습과 경험을 통해서만 익숙해질 수 있다. IDE를 사용 중이라면 처음엔 터미널 기반의 디버거가 익숙하지 않겠지만 계속해라.

디버거를 사용하는 다른 방법

디버거를 실행하는 몇 가지 다른 유용한 방법이 있다. set_trace 함수(pdb.set_trace 에서 유래한 이름) 이용하는 것이 그중 하나인데, '가난뱅이의 브레이크포인트'라고 불리는 것이다.

ㅏ래에 짧은 예제가 있는데 이 코드를 범용적으로 사용하기 위해 IPython 프로파일에 추가해서 사용하고 있다.