

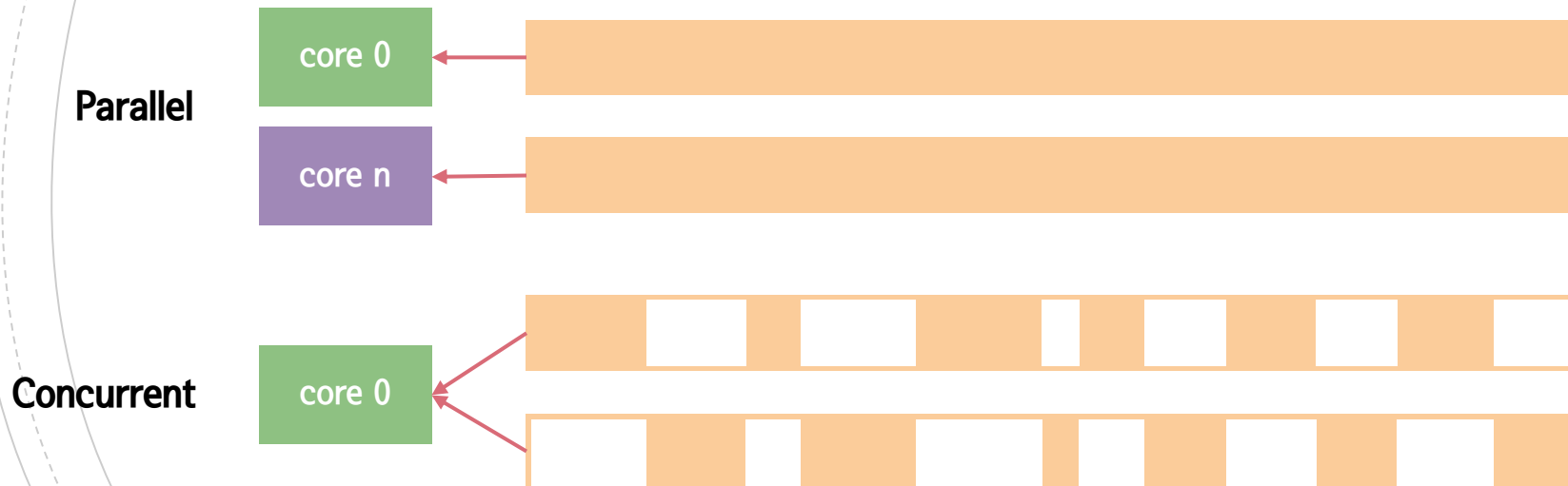
The background features a series of concentric circles in light gray, some solid and some dashed, creating a ripple effect. A solid purple rectangle is positioned horizontally in the upper-middle section of the image. Below the rectangle, a small purple triangle points downwards.

멀티 프로세싱

동시성과 병렬성

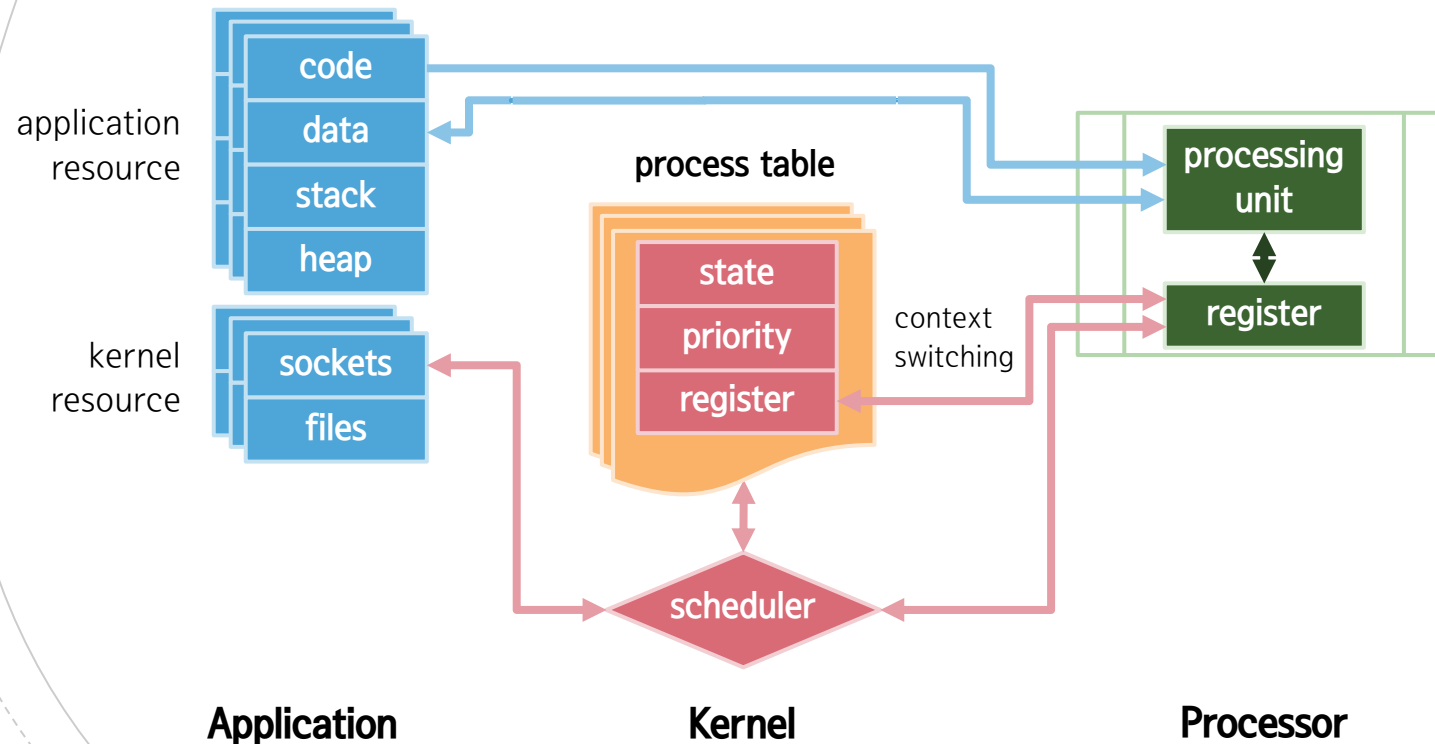
■ 동시성과 병렬성

- 동시성은 흔히 말하는 멀티태스킹으로 하나의 연산장치를 공유해 빠른 속도로 번갈아 가며 여러 작업 수행
 - 단순한 I/O 작업은 연산장치의 개입이 적으므로 대부분의 시간을 대기하면서 보냄 --> 동시성이 효율적
- 병렬성은 여러 개의 연산장치를 이용해 여러 작업을 각각의 연산장치에 동시에 수행
 - 물리적인 연산장치에 제약이 있음



다중 응용프로그램 실행 환경

- 커널은 스케줄러를 통해 여러 응용프로그램의 실행 관리
 - 동시에 여러 응용프로그램을 실행하기 위해 응용프로그램의 작업 흐름을 프로세스의 문맥 context 으로 관리
 - 프로세서에서 현재 실행 중인 작업은 레지스터를 통해 관리되며 **문맥은 레지터 값 모음**
 - 스케줄러는 프로세스 테이블을 통해 우선순위와 상태로 실행할 작업의 문맥을 저장 및 복원 ➡ 문맥 교환 context switching

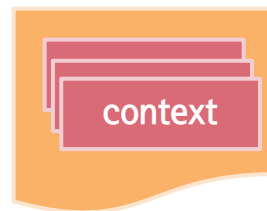


다중 응용프로그램 실행 환경

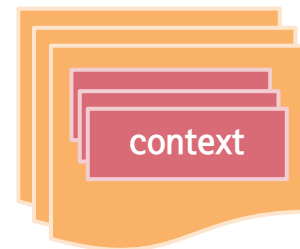
- 프로세스 단위로 문맥 관리 (멀티 프로세스)
 - 커널이 하나의 응용프로그램에 서로 다른 프로세스를 연관시키면 각 프로세스마다 다른 작업 흐름을 가질 수 있음 ➡ 프로세스 복제
 - 복제된 프로세스에 공유메모리 같은 커널 자원이 있으면 이를 공유하고, 전역변수나 스택, 힙 같은 응용프로그램 자원은 개별적임
 - 프로세스가 커널 자원에 접근할 때 문제가 발생하지 않도록 신경써야 함
 - 프로세스 사이 데이터 공유는 커널 자원 이용
 - 복제된 프로세스가 실행할 작업을 구분해 프로그램 작성
- 프로세스에 속한 작업 단위로 문맥 관리 (멀티 스레드)
 - 커널이나 응용라이브러리를 통해 하나의 응용프로그램이 동시에 실행한 여러 여러 작업을 가질 수 있도록 지원 ➡ 스레드 생성
 - 프로세스는 하나이므로 응용프로그램 자원 역시 공유
 - 여러 스레드가 동시에 커널 및 응용프로그램 자원에 접근할 때 문제가 발생하지 않도록 신경써야 함
 - 스레드 사이 데이터 공유는 커널 자원 외에 전역변수와 같은 응용프로그램 자원도 이용할 수 있음
 - 스레드로 실행할 작업을 구분해 프로그램 작성



multi-process
single-thread



single-process
multi-thread



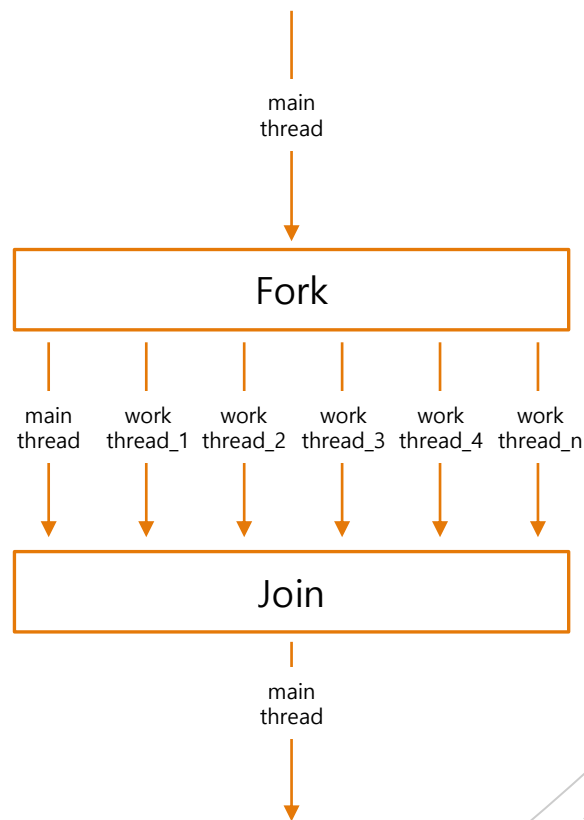
multi-process
multi-thread

threading 모듈

■ threading 모듈로 멀티 스레드 지원

- 하나의 프로세스 안에 모든 자원의 잠금을 전역으로 관리하여 한번에 하나의 스레드만 자원을 점유하며 동작
 - 여러 스레드를 동시에 실행시키지만, 결국 GIL 때문에 한번에 하나의 스레드만 계산 실행
- GIL로 쓰레기 수집기와 같은 자원 관리에 도움이 되지만 멀티 코어 지원은 부족
 - I/O 보다 연산이 중요한 작업에서는 스레드보다는 multiprocessing 모듈 권장
- 메인 스레드가 종료할 때 실행 중인 작업 스레드가 있으면 대기 상태가 됨 ➡ Join

```
01: import threading
02: import time
03:
04: def work_thread():
05:     print("work thread start", threading.currentThread().getName())
06:     time.sleep(5)
07:     print("work thread end", threading.currentThread().getName())
08:
09: print("main thread start")
10: for i in range(5):
11:     t = threading.Thread(target=work_thread);
12:     t.start()
13: print("main thread end")
```



명시적 Join

- 메인 스레드가 **작업 스레드를 생성하는 것을 Fork**라 하고 **작업 스레드의 종료를 대기하는 것을 Join**이라 함
 - 5개의 작업 스레드를 만들면 총 메인 스레드를 포함해 총 6개의 스레드가 스케줄링(CPU 자원 할당 가능 상태)됨.
 - 명시적 Join은 특정 작업 스레드가 작업을 마칠 때까지 사용자가 지정한 위치에서 명시적으로 대기하는 것을 의미
 - 여러 스레드의 작업 결과를 모아 최종 결과를 도출할 때 사용

```
01: import threading
02: import time
03:
04: def work_thread():
05:     print("work thread start", threading.currentThread().getName())
06:     time.sleep(5)
07:     print("work thread end", threading.currentThread().getName())
08:
09: print("main thread start")
10: for i in range(5):
11:     t = threading.Thread(target=work_thread);
12:     t.start()
13:     t.join()
14: print("main thread end")
```

데몬 스레드

- 메인 스레드가 종료될 때 작업 스레드도 즉시 종료
 - daemon 속성에 True 대입.

```
01: import threading
02: import time
03:
04: def work_thread():
05:     print("work thread start", threading.currentThread().getName())
06:     time.sleep(5)
07:     print("work thread end", threading.currentThread().getName())
08:
09: print("main thread start")
10: for i in range(5):
11:     t = threading.Thread(target=work_thread);
12:     t.daemon = True
13:     t.start()
14: print("main thread end")
```

스레드 인자

- 스레드는 함수처럼 전역 변수를 공유하지만 지역 변수처럼 인자 전달도 가능
 - 주의) args 인자에는 반드시 튜플 객체를 전달해야 함. --> 인자가 1개일 때 주의

```
01: import threading
02: import time
03:
04: def work_thread(name):
05:     print("work thread start", name)
06:     time.sleep(5)
07:     print("work thread end", name)
08:
09: print("main thread start")
10: for i in range(5):
11:     t = threading.Thread(target=work_thread, args=("%d"%(i+10),));
12:     t.start()
13: print("main thread end")
```

공유 데이터 접근과 동기화

- 여러 개의 작업 스레드가 전역 변수를 동시에 접근해 쓰기를 수행하면 데이터의 무결성이 깨질 수 있음
 - 뮤텝스와 같은 동기화 객체 사용

```
01: import threading
02:
03: count = 0
04: lock = threading.Lock()
05: t = [0 for i in range(5)]
06:
07: def work_thread(n):
08:     global count
09:
10:     lock.acquire()    #다른 스레드 접근 잠금
11:     for i in range(n):
12:         count += 1
13:     lock.release()    #다른 스레드 접근 잠금 해제
14:
13: for i in range(5):
14:     t[i] = threading.Thread(target=work_thread, args=(1000000,));
15:     t[i].start()
16:
17: for i in range(5):
18:     t[i].join()
19:
20: print("count = %d"%(count))
```

multiprocessing 모듈

- 파이썬의 GIL 문제를 회피하려면 multiprocessing 모듈로 멀티 프로세스 생성
 - 멀티 프로세스가 필요할 때 코어 수에 맞게 프로세스를 생성하면 실행 효율이 최적화됨
 - 부모(또는 메인) 프로세스는 파이썬 인터프리터
 - 부모 프로세스가 운영체제에 요청하여 자식 프로세스를 새로 만드는 과정을 스폰닝(spawning)이라 함
 - 부모 프로세스가 처리할 연산이 많은 경우 자식 프로세스를 만들어 일부 연산을 자식 프로세스에게 위임

```
01: import multiprocessing
02: import time
03:
04: def child_process():
05:     proc = multiprocessing.current_process()
06:     print(proc.name)
07:     print(proc.pid)
08:     time.sleep(5)
09:
10: print("parent process start")
11: for i in range(5):
12:     p = multiprocessing.Process(target=child_process)
13:     p.start()
14: print("parent process end")
```

프로세스 Pool과 결과 반환

- Pool은 지정된 개수만큼 자식 프로세스를 미리 만들어 놓고, 그 프로세스들 위에서 작업 수행
 - 코어 개수는 `cpu_count()` 함수로 파악
 - `apply_async()` 메소드로 Pool에 자식 프로세스를 넣고 실행
 - 필요할 때 반환값을 읽을 수 있도록 `AsyncResult` 객체 반환
 - 자식 프로세스의 반환 결과는 `AsyncResult` 객체의 `get()` 메소드로 읽음

```
01: import multiprocessing
02: import time
03:
04: def child_process(n):
05:     count = 0
06:
07:     for i in range(1, n+1):
08:         count += i
09:
10:     return count
11:
12: print("parent process start")
13: p = multiprocessing.Pool(multiprocessing.cpu_count())
14:
```

```
15: async_result = [None for i in range(5)]
16: result = [0 for i in range(5)]
17:
18: for i in range(5):
19:     async_result[i] = p.apply_async(child_process, (100,))
20:
21: for i in range(5):
22:     result[i] = async_result[i].get() #반환값을 받을 때까지 대기
23:
24: p.close() #Pool 자원 해제. (더이상 프로세스를 넣지 않음)
25: p.join() #자식 프로세스 종료 대기
26:
27: print("parent process end")
28: print(result, sum(result))
```



클로저를 일회성 또는 무한 스레드로 실행하는 클래스를 구현하시오.

- 사용자 함수는 클로저가 포함된 고차함수로 정의
 - 공통된 스레드 함수 호출을 위해 클로저의 인자는 고차함수의 자유변수로 한정
- threading 모듈의 Thread를 상속한 PopThread 클래스 정의
 - `__init__()` 메소드는 다음 속성 초기화
 - 인자로 전달받은 고차함수와 클로저의 자유변수 초기화
 - 실행 방법(일회성, 무한)과 실행 방법이 무한일 때 강제 종료를 위한 속성 초기화
 - `start()` 메소드로 스레드가 실행할 때 호출되는 `run()` 메소드는 재정의
 - 속성으로 저장한 고차 함수와 인자를 이용해 클로저를 얻음
 - 실행이 일회성이라면 클로저 호출, 아니면 종료 속성이 False인 동안 while 루프로 클로저 호출 반복
 - 스레드 실행 상태를 반환하는 `state()` 메소드 정의
 - 종료 속성 반환
 - 스레드가 무한 실행일 때 안전하게 종료하는 `stop()` 메소드 정의
 - 종료 속성을 True로 설정



클로저를 일회성 또는 무한 스레드로 실행하는 클래스를 구현하시오.

■ popthread.py

```
01: from threading import Thread
02: import time
03:
04: class PopThread(Thread):
05:     def __init__(self, func, once=True, *args, **kwargs):
06:         Thread.__init__(self)
07:         self.daemon = True
08:         self.func = func
09:         self.once = once
10:         self.args = args
11:         self.kwargs = kwargs
12:         self.end = False
13:
14:     def run(self):
15:         work = self.func(*self.args, **self.kwargs)
16:
17:         if self.once:
18:             work()
19:             self.end = True
20:         else:
21:             while not self.end:
22:                 work()
23:
```

```
24:     def state(self):
25:         return not self.end
26:
27:     def stop(self):
28:         self.end = True
29:
```



- popthread_main.py

```
01: from popthread import PopThread
02:
03: def log(msg):
04:     def wrapper():
05:         print(msg + "Run")
06:         time.sleep(0.01)
07:     return wrapper
08:
09: def trace(msg):
10:     def wrapper():
11:         print(msg + "Run")
12:     return wrapper
13:
14: t1 = PopThread(trace, True, "### TRACE:")
15: t2 = PopThread(log, False, ">>> LOG:")
16:
17: t1.start()
18: t2.start()
19:
20: for i in range(10):
21:     print("+++ MAIN: Run")
22:     time.sleep(0.01)
23:
24: print("+++ MAIN: End")
```

[illegible]

NumPy와 matplotlib 라이브러리

[NumPy]

NumPy

- 강력한 N차원 배열 객체로 범용적 데이터 처리를 위한 다차원 컨테이너
 - 과학 연산을 위한 파이썬 핵심 라이브러리 중 하나로 빠른 고성능 연산을 위해 C언어로 구현
 - 파이썬의 편의성과 C언어의 연산 능력을 동시에 이용
 - 벡터부터 텍서에 이르기까지 다양한 방법으로 만드는 다차원 배열 제공하며, 선형대수 문제를 쉽게 처리할 수 있음
 - 스칼라 scalar
 - 하나의 값. $a = 10$
 - 벡터 vector
 - 순서가 있는 1차원 배열. $x = [0, 1, 2]$
 - 순서가 없는 배열은 집합 set
 - 행렬 matrix
 - 벡터 m 이 n 개 존재($m \times n$)하는 2차원 배열
 - $1 \times n$ 행 row 벡터 $[[1 \ 2]]$ 와 $m \times 1$ 열 column 벡터 $\begin{bmatrix} 1 \\ 2 \end{bmatrix}$ 는 서로 전치 관계
 - 텐서 tensor
 - 같은 크기의 행렬로 구성된 3차원 이상 배열
 - 인기있는 서드파티 라이브러리들이 NumPy를 기본 자료구조로 사용하거나 호환됨
 - matplotlib, pandas, opencv, pytorch, tensorflow 등

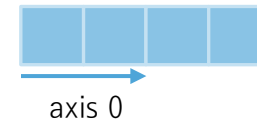
ndarray

■ NumPy의 다차원 배열 객체

• ndarray의 주요 속성

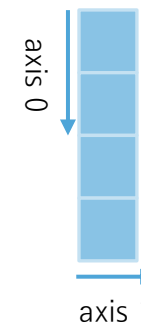
- ndarray.ndim: 배열의 축 axis (차원 dimensions) 수
- ndarray.shape: 배열의 차원으로, 각 차원 배열의 크기를 정수 튜플로 나타냄.
 - (4_{axis 0}), (4_{axis 0}, 3_{axis 1}), (2_{axis 0}, 4_{axis 1}, 3_{axis 2})
 - 4 ≡ (4,) ➡ (1, 4): 1x4 행 벡터
 - (4, 1): 4x1 열 벡터
- ndarray.size: 배열의 총 요소 수로, 각 차원 배열의 크기를 모두 곱한 값
- ndarray.dtype: 배열 요소 타입
 - 정수: numpy.int8, numpy.int16, numpy.int32, numpy.int64 (== int, 정수 기본 타입)
 - 실수: numpy.float16, numpy.float32, numpy.float64 (== float, 실수 기본 타입), numpy.float128
 - 기타: numpy.complex, numpy.bool, numpy.str, numpy.object
- ndarray.itemsize: 바이트 단위 배열 요소의 크기 (요소 타입 크기)
- ndarray.data: 실제 배열 요소를 포함하는 버퍼로, 이미지나 오디오 같은 바이너리 데이터를 다룰 때 사용

1D array



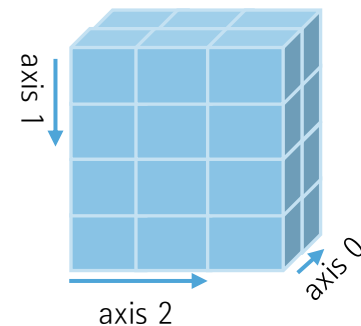
ndim: 1
shape: (4,)
size: 4
dtype: int64
itemsize: 8

2D array



ndim: 2
shape: (4, 1)
size: 4
dtype: int64
itemsize: 8

3D array



ndim: 3
shape: (2, 2, 2)
size: 8
dtype: int64
itemsize: 8

배열 생성

- 파이썬 객체 (리스트, 튜플)
 - 2차원 이상은 반드시 열의 개수가 일치해야 함
 - 요소 중에 실수가 하나라도 포함되면 실수 타입
 - 정수 요소의 디폴트 타입은 int64이고 실수 요소의 디폴트 타입은 float64

```
01: import numpy as np
02:
03: show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
04:
05: a1 = np.array([2, 3, 4], dtype=np.int8)
06: a2 = np.array((1.5, 7, 9, 10))
07: a3 = np.array([[2, 3, 4], [7, 9, 10]])
08:
09: show("a1:", a1)
10: show("a2:", a2)
11: show("a3:", a3)
```



배열 생성

■ 범위 기반

- arange()는 range()처럼 주어진 [start, stop)에서 균일한 정수 또는 실수 배열 반환
 - numpy.arange([start]stop, [step,]dtype=None)
- linspace()는 arange()와 유사하나 step이 실수이면 오차가 발생할 수 있으므로 개수 사용
 - numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)

```
01: import numpy as np
02:
03: show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
04:
05: a1 = np.arange(27+1)
06: a2 = np.arange(1, 10+1, 0.5)
07: a3 = np.linspace(1, 10+1, 20)
08:
09: show("a1:", a1)
10: show("a2:", a2)
11: show("a3:", a3)
```



배열 생성

■ 특정 값으로 초기화

- `numpy.zeros(shape, dtype=float)`: 배열 전체를 0으로 초기화
- `numpy.ones(shape, dtype=None)`: 배열 전체를 1로 초기화. 기본 타입은 float
- `numpy.full(shape, fill_value, dtype=None)`: 배열 전체를 전달한 값으로 초기화
- `numpy.eye(N, M=None, k=0, dtype=float)`: $N \times M$ 배열에 대해 대각선이 1이고 나머지는 0으로 초기화
 - M을 생략하면 $M=N$ (항등 또는 단위 행렬 identity matrix), k는 대각선 인덱스로 0은 기준 대각, 양수는 기준 대각 위쪽, 음수는 아래쪽
- `numpy.diag(a, k=0)`: 단위 행렬의 대각선 값들을 하나의 배열로 반환

```
01: import numpy as np
02:
03: show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
04:
05: a0 = np.zeros(5)
06: a1 = np.zeros((5,1), dtype=int)
07: a2 = np.ones((3, 4), dtype=int)
08: a3 = np.full((2, 3), -1)
09: a4 = np.eye(5, k=1)
10: a5 = np.diag(a4, k=1)
11:
12: show("a0:", a0) ; show("a1:", a1) ; show("a2:", a2)
13: show("a3:", a3) ; show("a4:", a4) ; show("a5:", a5)
```



배열 생성

■ 기존 배열 복사

- `numpy.ones_like(a, dtype=None)`: 1로 채워진 배열
- `numpy.zeros_like(a, dtype=None)`: 0으로 채워진 배열
- `numpy.full_like(a, fill_value, dtype=None)`: 전달한 값으로 채워진 배열

```
01: import numpy as np
02:
03: show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
04:
05: a = np.linspace(1, 12, 12).reshape((2, 3, 2))
06:
07: b1 = np.ones_like(a)
08: b2 = np.zeros_like(a)
09: b3 = np.full_like(a, -1)
10:
11: show("a:", a)
12: show("b1:", b1)
13: show("b2:", b2)
14: show("b3:", b3)
```

모양 변경

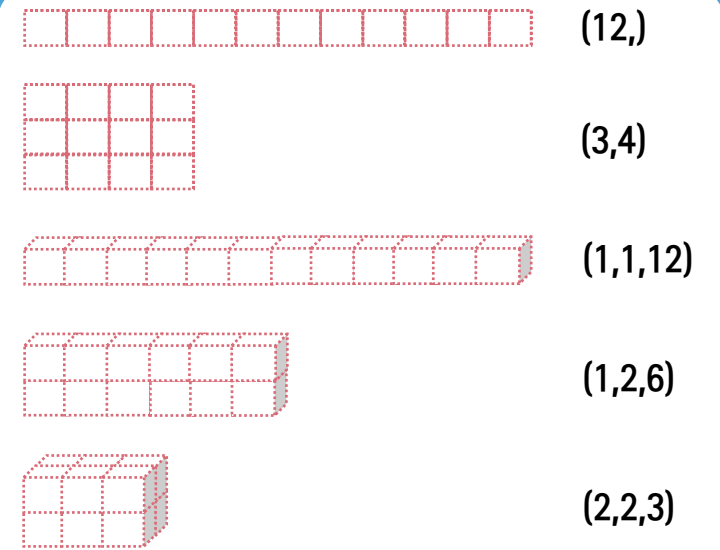
- 배열은 각 축을 따라 요소의 수로 지정된 모양을 가짐
 - 배열 자체는 변하지 않고 새로운 차원으로 모양만 바꾸므로 size는 같아야 함
 - `numpy.reshape(a, new_shape)`
 - `ndarray.reshape(new_shape)`
 - 모양에 맞춰 배열 크기도 함께 변경하므로 새 모양의 size가 더 크면 나머지 요소는 0으로 채움
 - `ndarray.resize(new_shape)`

```
01: import numpy as np
02:
03: show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
04:
05: a1 = np.full((6, 3), -1)
06: a2 = np.reshape(a1, (3, 3, 2))
07: a3 = np.arange(1, 10 + 1, 0.5).reshape((5, 4))
08: a4 = np.linspace(1, 10 + 1, 20).reshape((2, 5, 2))
09:
10: show("a1:", a1)
11: show("a2:", a2)
12: show("a3:", a3)
13: show("a4:", a4)
```

memory (12 elements)



shape



기본 연산

■ 산술 연산과 관계 연산

- 산술 연산과 관계 연산은 **행렬 요소 사이 1:1로 적용**되며 관계 연산은 불 결
 - 피 연산자는 배열 또는 스칼라 값으로 둘 다 배열일 때는 shape와 size가 같아야 함
 - 스칼라 값은 같은 크기 및 모양의 배열로 브로드캐스팅 broadcasting 한 후 연산 수행
 - 산술 연산은 연산자 (+, -, *, /, %, **)와 함수(numpy 모듈의 add, subtract, multiply, divide, mod, power) 모두 지원. *는 일반적인 행렬 곱셈이 아님
 - 관계 연산의 결과는 불 배열

```
01: import numpy as np
02:
03: show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
04:
05: a1 = np.array([10, 20, 30, 40, 50, 60])
06: a2 = np.linspace(5, 6, a1.size)
07:
08: b1 = a1 - a2
09: b2 = np.power(b1, 2)   # b1 ** [2, 2, 2, 2, 2, 2]
10: b3 = np.sin(a1) * 10   # np.sin(a1) * [10, 10, 10, 10, 10, 10]
11: b4 = a1 % a2
12: b5 = b1 < 25           # b1 < [25, 25, 25, 25, 25, 25]
13:
14: show("a1:", a1) ; show("a2:", a2)
15: show("b1:", b1) ; show("b2:", b2) ; show("b3:", b3) ; show("b4:", b4) ; show("b5:", b5)
```

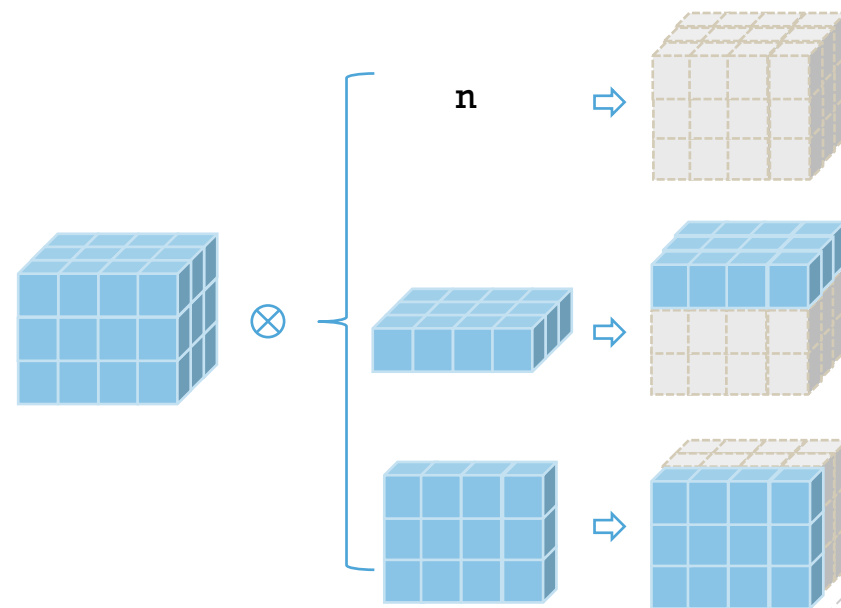


기본 연산

■ 브로드캐스팅

- 스칼라 값이나 벡터를 배열과 연산할 때 이를 배열의 share와 같도록 확장한 후 기존 데이터 복사
 - 배열과 스칼라 값 사이 연산: 스칼라를 배열의 share로 확장한 후 스칼라 값 복사
 - 벡터와 벡터 사이 연산: 벡터 N, M에 대해 양쪽 다 배열의 N x M shape로 확장한 후 행 또는 열 단위 요소 복사
 - 배열과 벡터 사이 연산: **배열의 마지막 축 크기와 벡터의 크기가 같을 때**, 벡터를 배열의 shape로 확장한 후 벡터 요소 복사
 - 크기가 다른 배열과 배열 사이 연산: 두 배열의 마지막 축부터 차례로 비교해 **축의 크기가 같거나 1일 때**, 양쪽 배열을 큰 축 기준으로 확장한 후 배열 요소 복사

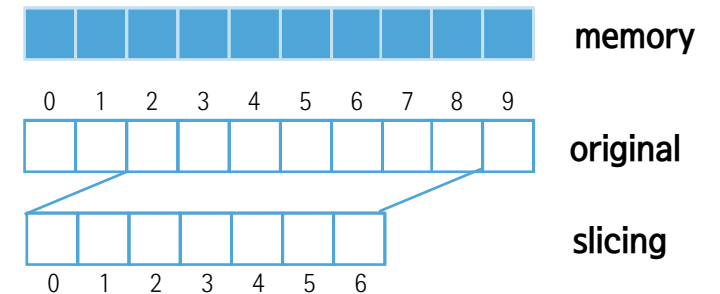
```
01: import numpy as np
02:
03: show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
04:
05: a1 = np.linspace(1, 12, 12).reshape((3, 4))
06: a2 = np.arange(1, 5)
07: a3 = a2.reshape((4, 1))
08: a4 = a1.reshape((3, 1, 4))
09:
10: b1 = a1 + a2
11: b2 = a2 + a3
12: b3 = a1 + a4
13:
14: show("a1:", a1) ; show("a2:", a2) ; show("a3:", a3)
15: show("b1:", b1) ; show("b2:", b2) ; show("b3:", b3)
```



인덱싱과 슬라이싱

- 리스트처럼 NumPy 배열도 인덱스로 요소에 접근하고 원하는 부분만 잘라내는 슬라이싱 가능
 - 리스트와 다른점은 **슬라이싱은 원본 배열의 데이터를 참조하는 새로운 배열**이므로 **슬라이싱된 배열을 수정하면 원본 배열도 함께 수정됨**
 - 1차원 배열의 슬라이싱은 `[[start]:[end]:[step]]`
 - 인덱스 i 가 start일 때 $i < \text{end}$ 동안 $i += \text{step}$ 단위로 이동하며 요소 접근. i 가 음수면 마지막 요소부터 역순으로 접근
 - start를 생략하면 0, end를 생략하면 -1(마지막 요소), step을 생략하면 1

```
01: import numpy as np
02:
03: show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
04:
05: a1 = np.arange(1, 10+1)**2
06: a2 = a1[2:9]
07:
08: show("a1", a1) ; show("a2", a2)
09:
10: a1[3] = a1[1] + a1[2]
11: a2[:5:2] = 10_1000
12:
13: for i in range(len(a1)):
14:     print(a1[(i+1)*-1], end=',')
15: print()
```

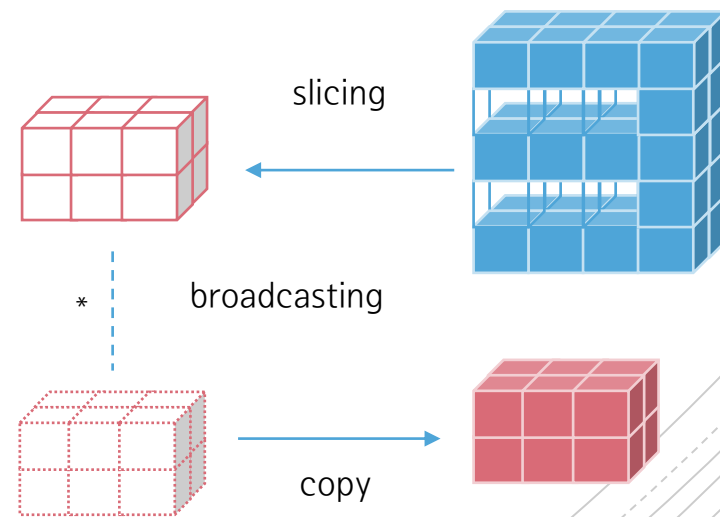




인덱싱과 슬라이싱

- 다차원 배열은 축당 하나의 인덱스를 가질 수 있으며 인덱스는 쉼표로 구분 ➡ [x, y]
 - 축 수보다 적은 인덱스를 제공하면 누락된 인덱스는 슬라이스로 간주
 - 점(...)은 완전한 인덱싱에 필요한 만큼의 콜론을 나타냄
 - 5차원 배열 x에 대해
 - $x[1, 2, \dots] == x[1, 2, :, :, :]$
 - $x[\dots, 3] == x[:, :, :, :, 3]$
 - $x[4, \dots, 5, :] = x[4, :, :, 5, :]$
 - 다차원 배열에 대한 반복은 첫 번째 축에 대해 수행
 - 반복자인 flat 속성을 이용하면 배열의 모든 요소에 접근할 수 있음
- 슬라이싱할 때 추가 연산을 통해 **브로드캐스팅이 발생하면 사본 복사**

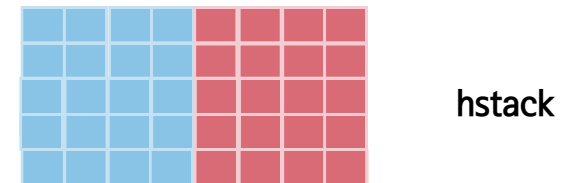
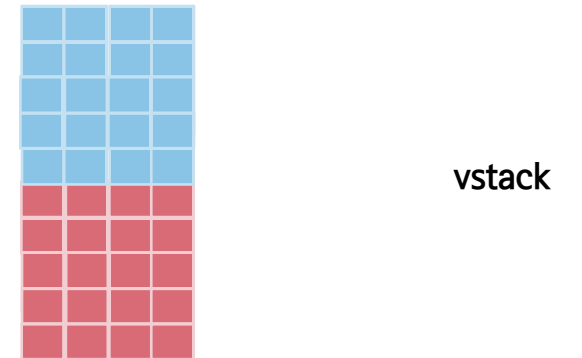
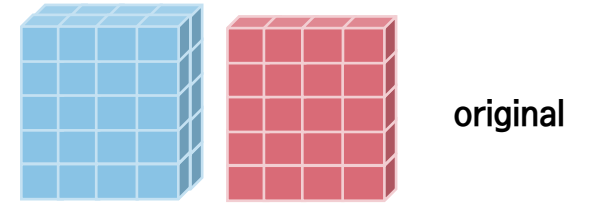
```
01: import numpy as np
02:
03: show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
04:
05: a1 = np.fromfunction(lambda x, y, z: x + y + z, (2, 5, 4), dtype=int)
06: a2 = a1[:, 1::2, :3] * 10          # a2는 새로운 배열
07:
08: a2[1, ...] = -1
09:
10: show("a1", a1)
11: for array in a2:                  # for element in a2.flat
12:     print("out:\n", array)        #     print(element, end=', ')
```



서로 다른 배열 쌓기

- 서로 다른 축을 따라 여러 배열을 수평 또는 수직으로 쌓은 새로운 배열 생성
 - `vstack(tup)`, `hstack(tub)`: 세로(행 방향), 가로(열 방향) 순서로 쌓음
 - `tup`: 배열 시퀀스로 배열 모양은 첫 번째 축을 제외하고 모두 동일해야 함. 1차원 배열은 길이가 같아야 함
 - `concatenate((a1, a2, ...), axis=0, ...)`: 기존 축을 따라 배열 시퀀스 결합
 - `a1, a2, ...`: 배열 시퀀스로 첫 번째 축의 차원을 제외하고 동일한 모양을 가져야 함
 - `axis`: 배열이 결합될 축. `None`으로 설정하면 1차원으로 병합

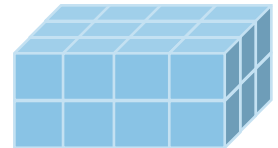
```
01: import numpy as np
02:
03: show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
04:
05: a1 = np.fromfunction(lambda x, y, z: x + y + z, (2, 5, 4), dtype=int)
06: a2 = np.arange(1, (1*5*4)+1).reshape((1, 5, 4)) * 10
07:
08: a3 = np.vstack((a1[0, ...], a2[0, ...]))
09: a4 = np.hstack((a1[1, ...], a2[0, ...]))
10: a5 = np.concatenate((a1, a2), 0)
11:
12: show("a1", a1) ; show("a2", a2)
13: show("a3", a3) ; show("a4", a4) ; show("a5", a5)
```



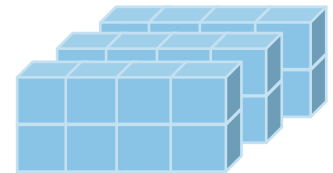
작은 배열로 분할

- 하나의 배열은 여러 개의 작은 배열로 분할될 수 있음
 - `vsplit(ary, indices_or_sections)`: 배열을 세로로(행 방향) 첫 번째 축을 따라 여러 하위 배열로 분할
 - `ary`는 배열, `indices_or_sections`는 분할 개수 또는 분할 위치 지정 스퀀스 또는 배열
 - 반환은 튜플 배열
 - `hsplit(ary, indices_or_sections)`: 배열을 가로로(열 방향) 두 번째 축을 따라 여러 하위 배열로 분할

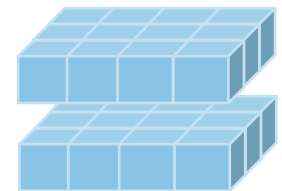
```
01: import numpy as np
02:
03: show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
04:
05: a = np.arange(24).reshape(3, 2, 4)
06:
07: b1, b2, b3 = np.vsplit(a, 3)
08: b4, b5 = np.vsplit(a, np.array([1]))
09:
10: c1, c2 = np.hsplit(a, 2)
11:
12: show("a", a)
13: show("b1", b1) ; show("b2", b2) ; show("b3", b3) ; show("b4", b4) ; show("b5", b5)
14: show("c1", c1) ; show("c2", c2)
```



original



vsplit (axis=0)



hsplit (axis=1)

참조와 복사

- 배열에 대한 연산 결과로 만들어지는 새로운 배열은 기존 배열의 참조 또는 복사
 - 대입문에 의한 단순 할당과 함수의 인자 전달은 객체 참조
 - 얇은 복사: view() 메소드나 슬라이싱은 동일한 데이터를 새로운 관점에서 보여주는 새로운 배열 객체를 만듦
 - 깊은 복사: copy() 메소드는 배열과 해당 데이터의 전체 사본을 만듦

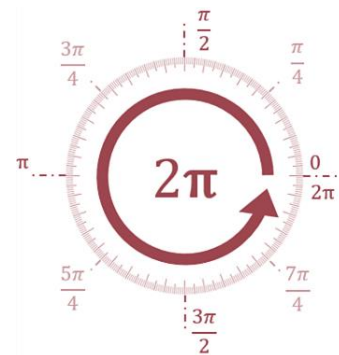
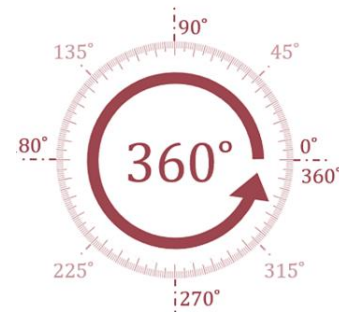
```
01: import numpy as np
02:
03: show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
04:
05: def foo(n):
06:     return id(n) # id()는 변수에 저장된 참조 값 반환
07:
08: a = np.arange(20).reshape(4, 5)
09: b = a
10: c = a.view()
11:
12: print(b is a, id(a) == foo(a))
13: print(c is a, c.flags.owndata) # .flags.owndata 데이터 유무
14:
```

```
15: d = c.reshape((2, 10))
16: d[1, 0] = 1_000_000
16: show("d", d) ; show("a", a)
17:
18: s = a[1:3]
19: s[:] = -1
20: show("a", a)
21:
22: e = a.copy()
23: print(e is a)
24: d[1, 2] = 2_000_000
25: show("e", e) ; show("a", a)
26:
27: m = np.arange(1_000_000)
28: n = m[:100].copy()
29: del m
```

수학 함수

■ 삼각 함수

- `numpy.pi`: π 상수
- `numpy.sin(x)`, `numpy.cos(x)`, `numpy.tan(x)` : x 에 대해 sine, cosine, tangent 계산. x 는 라디안 값 또는 배열
- `numpy.arcsin(x)`, `numpy.arccos(x)`, `numpy.arctan(x)`: x 에 대해 sine, cosine, tangent의 역 함수 계산
- `numpy.radians(x)`: x 에 대해 디그리를 라디안으로 변환. $degree(\frac{\pi}{180})$
- `numpy.degrees(x)`: x 에 대해 라디안을 디그리로 변환. $radian(\frac{180}{\pi})$



```
01: import numpy as np
02:
03: show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
04:
05: print(np.pi)
06: print(np.sin(30 * np.pi / 180), np.cos(3 * np.pi / 2), end='\n\n')
07:
08: a1 = np.degrees([np.pi/4, 3*np.pi/4, 5*np.pi/4, 7*np.pi/4])
09: a2 = np.radians(a1)
10: a3 = np.sin(a1 * np.pi / 180)
11: a4 = np.cos(a2)
12:
13: show("a1", a1) ; show("a2", a2) ; show("a3", a3) ; show("a4", a4)
```

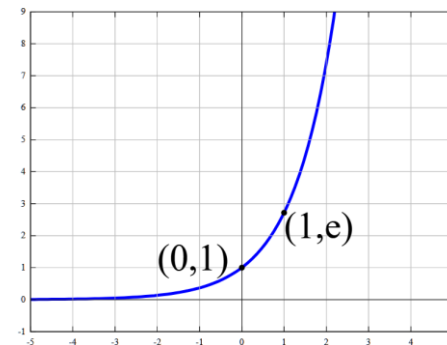


수학 함수

■ 지수와 로그

- `numpy.e` : 지수(자연 로그 밑) 상수(약 2.718281...)
- `numpy.exp(x)`: x 에 대해 $y = e^x$ 인 지수(자연 로그 역) 계산
- `numpy.log(x)`: x 에 대해 밑이 e 인 자연 로그(지수 함수 역) 계산
 - 밑이 10인 상용로그는 `numpy.log10(x)`

x	$(1+x)^{\frac{1}{x}}$	x	$(1+x)^{\frac{1}{x}}$
0,1	2,59374...	-0,1	2,86797...
0,01	2,70481...	-0,01	2,73199...
0,001	2,71692...	-0,001	2,71964...
0,0001	2,71814...	-0,0001	2,71841...
0,00001	2,71826...	-0,00001	2,71829...
⋮	⋮	⋮	⋮



```
01: import numpy as np
02:
03: show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
04:
05: d1 = np.exp(1)
06: a1 = np.exp([i for i in range(1, 10 + 1, 2)])
07: d2 = np.log(np.e)
08: a2 = np.log(a1)
09:
10: print("d1:", d1)
11: print("d2:", d2)
12:
13: show("a1", a1)
14: show("a2", a2)
```



수학 함수

■ 기타 함수

- `numpy.abs(x)`, `numpy.absolute(x)`: `x`에 대해 절대값 계산
 - `abs()`는 정수 한정
- `numpy.ceil(x)`: `x`에 대해 `x`보다 작은 정수 중 가장 큰 값 계산
- `numpy.floor(x)`: `x`에 대해 `x`보다 큰 정수 중 가장 작은 값 계산
- `numpy.sqrt(x)`: `x`에 대해 제곱근 계산

```
01: import numpy as np
02:
03: show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
04:
05: a1 = np.array([-2.5, 1, -1.7, 5.1, 4.0, -0.2, 6.8])
06: a2 = np.absolute(a1)
07: a3 = np.ceil(a1)
08: a4 = np.floor(a1)
09: a5 = np.sqrt(a2)
10:
11: show("a1", a1)
12: show("a2", a2)
13: show("a3", a3)
14: show("a4", a4)
15: show("a5", a5)
```

선형 대수

■ 전치 행렬 transposed matrix

- 행과 열을 서로 맞바꾼 행렬로 numpy 모듈이나 ndarray 객체의 transpose() 또는 T 프로퍼티 사용

$$A = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \Rightarrow A^T = \begin{pmatrix} a & d & g \\ b & e & h \\ c & f & i \end{pmatrix}$$

```
01: import numpy as np
02:
03: show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
04:
05: a1 = np.array([1, 2, 3, 4])
06: a2 = np.array([[1, 2, 3], [4, 5, 6]])
07: a3 = np.linspace(10, 120, 12).reshape((3, 2, 2))
08:
09: b1 = a1.T
10: b2 = a2.transpose()
11: b3 = np.transpose(a3)
12:
13: show("a1:", a1) ; show("a2:", a2) ; show("a3:", a3)
14: show("b1:", b1) ; show("b2:", b2) ; show("b3:", b3)
```



선형대수

■ 행렬 곱셈

- 두 행렬에 대한 곱셈은 연산자 @ 또는 numpy 모듈이나 ndarray 객체의 dot() 사용
 - 첫 번째 배열 행의 요소 수와 두 번째 배열 열의 요소 수는 같아야 함

$$\begin{aligned} ax + by + cz &= p \\ dx + ey + fz &= q \\ gx + hy + iz &= r \end{aligned} \iff \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

```
01: import numpy as np
02:
03: show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
04:
05: a1 = np.array([[2, -1, 5], [-5, 2, 2], [2, 1, 3]])
06:
07: a2 = np.array([[0, 1, 0], [1, 0, 1], [1, 1, 0]])
08: a3 = np.array([1, -1, 1])
09: a4 = np.array([1, 0, 1]).reshape((3,1))
10:
11: b1 = np.dot(a1, a2)
12: b2 = a1.dot(a3)
13: b3 = a1 @ a4
14:
15: show("a1:", a1) ; show("a2:", a2) ; show("a3:", a3) ; show("a4:", a4)
16: show("b1:", b1) ; show("b2:", b2) ; show("b3:", b3)
```



선형대수

■ 역 행렬과 방정식 해

- 역 행렬은 $(n \times n)$ 정방 행렬에 대한 곱셈 결과가 항등원인 단위행렬을 만드는 행렬

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad A^{-1} = \begin{bmatrix} x & y \\ u & v \end{bmatrix} \quad \Rightarrow \quad A A^{-1} = E \quad \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x & y \\ u & v \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

- `numpy.linalg.inv(a)`: 배열의 역 행렬 계산
- `numpy.linalg.solve(a, b)`: 연립방정식 해 계산

$$\begin{array}{l} 2x + 3y = 4 \\ 5x + 6y = 7 \end{array} \quad \Rightarrow \quad \begin{bmatrix} 2 & 3 \\ 5 & 6 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \end{bmatrix}$$

```
01: import numpy as np
02:
03: show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
04:
05: a1 = np.random.randint(10, size=(3, 3))
06: a2 = np.linalg.inv(a1)
07: a3 = np.array([[2, 3], [5, 6]])
08: a4 = np.array([4, 7])
09: a5 = np.linalg.solve(a3, a4)
10:
11: show("a1", a1) ; show("a2", a2)
12: show("a3", a3) ; show("a4", a4) ; show("a5", a5)
```



연립 방정식의 해를 역 행렬을 이용해 구하시오

- 연립 방정식에 대한 A, B 배열에 대해 $A^{-1} \cdot B$ 계산

$$\begin{array}{l} 2x + 3y = 4 \\ 5x + 6y = 7 \end{array} \Rightarrow \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 & 3 \\ 5 & 6 \end{bmatrix}^{-1} \begin{bmatrix} 4 \\ 7 \end{bmatrix}$$

```
01: import numpy as np
02:
03: show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
04:
05: A = np.array([[2, 3], [5, 6]])
06: B = np.array([4, 7])
07: C = np.linalg.inv (A)
08: D = np.dot(C, B)
10:
11: show("solve", D)
```

랜덤 샘플

- 균등 분포 uniform distribution : 확률 밀도 $\mathcal{P}(x) = \frac{1}{b-a}$ 에 대해 값이나 요소를 $[a, b)$ 에서 균등 배치
 - `numpy.random.random(size=None)`: $[0.0, 1.0)$ 범위 실수
 - `size`는 생략하거나 스칼라 값, 2차원 이상은 `shape`로 반환은 스칼라 값 또는 배열
 - `numpy.random.uniform(low=0.0, high=1.0, size=None)`: $[low, high)$ 범위 실수 타입
 - `numpy.random.randint(low, high=None, size=None, dtype=int)`: $[low, high)$ 범위 정수 타입
 - `high`를 생략하면 `low = 0, high = low`

```
01: import numpy as np
02:
03: show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
04:
05: a1 = np.random.random(6)
06: a2 = np.random.random((2, 3))
07: a3 = np.random.uniform(1.0, 2.0, (2, 3))
08: d = np.random.randint(10)
09: a4 = np.random.randint(1, 10, (2, 3))
10:
11: show("a1", a1) ; show("a2", a2) ; show("a3", a3)
12: print("d:", d) ; show("a4", a4)
```



랜덤 샘플

- `numpy.random.shuffle(x)`: 시퀀스 `x`의 내용을 섞어서 수정. 다차원 배열은 첫 번째 축 기준
- `numpy.random.choice(a, size=None, replace=True, p=None)`: 주어진 1차원 배열에서 무작위 샘플 생성
 - `replace`는 선택 값 재 사용 유무, `p`는 `a` 항목 선택 확률
- `numpy.seed(seed=None)`: 의사 난수 제너레이터에서 사용할 시드 값 설정

```
01: import numpy as np
02:
03: show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
04:
05: np.random.seed(1)
06:
07: a1 = np.arange(9)
08: np.random.shuffle(a1)
09:
10: a2 = np.arange(3*3).reshape((3, 3))
11: np.random.shuffle(a2)
12:
13: a3 = np.random.choice([4, 2, 6, 1], 3, False, [0.4, 0.2, 0.3, 0.1])
14:
15: show("a1", a1)
16: show("a2", a2)
17: show("a3", a3)
```



랜덤 샘플

■ 정규 분포 normal distribution

- 가우스 분포의 확률 밀도 $\mathcal{P}(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$ 에 대해 μ 는 평균, σ 는 표준편차, σ^2 는 분산
 - 평균에서 최대치이며, 표준 편차와 함께 확산되며 증가함
 - 멀리 떨어져 있는 것보다 평균에 가까운 샘플을 반환할 확률이 높음
- `numpy.random.normal(loc=0.0, scale=1.0, size=None)`: 정규 분포
 - `loc`은 분포의 평균(중앙), `scale`은 양의 값으로 표준 편차(확산 또는 너비), `size`는 스칼라 또는 shape
- `numpy.random.standard_normal(size=None)`: 표준 정규 분포 standard normal (평균=0, 표준편차=1). 무작위 샘플은 $N(\mu, \sigma^2)$

```
01: import numpy as np
02:
03: show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
04:
05: mu, sigma = 0, 0.1
06: a1 = np.random.normal(mu, sigma, 50)
07: a2 = np.random.normal(4, 1.7, size=(3, 4)) # N(4, 2.89)
08: a3 = np.random.standard_normal(50)
09: a4 = 4 + 1.7 * np.random.standard_normal((3,4)) # N(4, 2.89)
10:
11: show("a1:", a1) ;
12: print("mean and standard deviation:", abs(mu - np.mean(a1)), sigma - np.std(a1, ddof=1), end="\n\n")
13:
14: show("a2:", a2) ; show("a3", a3) ; show("a4", a4)
```

파일 저장 및 로드

- 데이터의 재 사용을 위해 ndarray의 배열 요소를 파일로 저장하거나 파일로부터 데이터를 읽어 ndarray 객체로 변환
 - 바이너리 형식으로 확장자는 .npy
 - `numpy.save(fname, arr)` : 1개 배열을 파일로 저장
 - file은 확장자 생략 가능
 - `numpy.savez(fname, *args, **kwargs)` : n개의 배열을 파일로 저장
 - kwargs는 배열 이름을 키, 배열을 값으로 나열한 키워드 인자
 - `numpy.load(fname)`: .npy 파일에서 배열 로드. 압축된 바이너리라면 압축 해제
 - 파일명은 확장자까지 모두 포함해야 하며, 반환 객체는 ndarray와 호환되는 `numpy.lib.npyio.NpzFile`
 - `numpy.lib.npyio.NpzFile.close()`: 파일 닫기
 - 파일을 닫으면 더 이상 배열 접근 불가
 - 압축된 바이너리 파일
 - `numpy.savez_compressed(fname, *args, **kwargs)`: 압축 후 저장
 - 텍스트 파일(엑셀 등과 호환) 형식
 - `numpy.savetxt(fname, x)`: 파일 저장
 - x는 1차원 또는 1차원 배열
 - `numpy.loadtxt(fname)`: 파일 로드



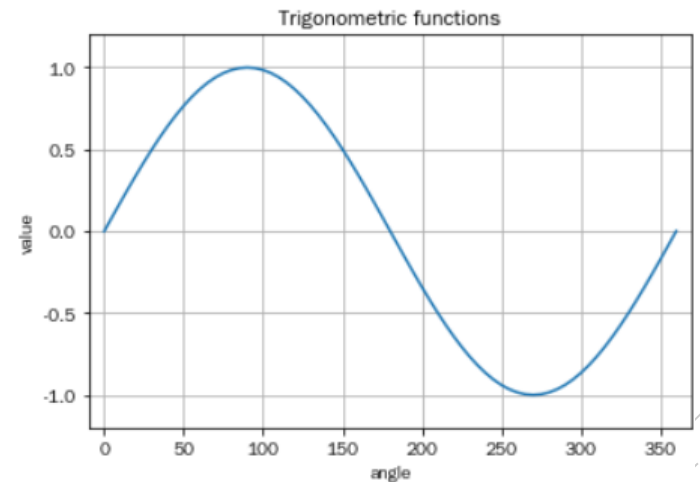
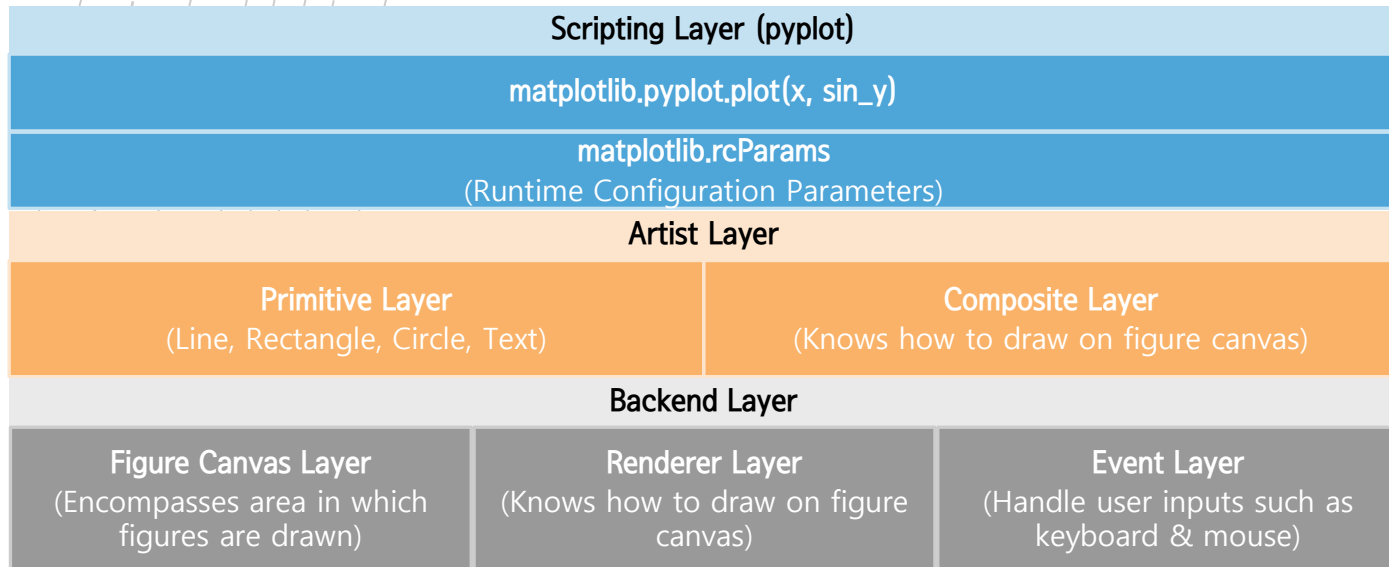
파일 저장 및 로드

```
01: import numpy as np
02:
03: show = lambda m, o : print(m, o.shape, o.dtype, '\n', o, '\n')
04:
05: a1 = np.arange(360+1)
06: a2 = np.sin(a1 * np.pi / 180)
07:
08: show("a1", a1)
09: show("a2", a2)
10:
11: np.savez_compressed("sin_sample", x=a1, y=a2)
12:
13: a3 = np.load("sin_sample.npz")
14:
15: print((a3['x'] == a1).all()) # a3['x'] == a1의 결과는 불 배열이며, all()은 전체 요소가 같은지 검사
16: print((a3['y'] == a2).all())
17:
18: a3.close()
```

NumPy와 matplotlib 라이브러리 [matplotlib]

matplotlib

- 둘 이상의 데이터 사이 관계를 플롯으로 나타내는 가장 오래된 파이썬 플로팅 plotting 라이브러리
 - MATLAB과 유사한 오픈소스 과학 컴퓨팅 라이브러리인 SciPy의 일부로 2003년 개발되어 현재까지 업데이트가 이어짐
 - 파이썬으로 작성되었으며 NumPy 및 기타 확장 코드를 사용하여 대규모 배열에서도 우수한 성능 제공
 - 아키텍처는 세 가지 주요 계층으로 구성됨
 - 스크립팅 scripting , 아티스트 artist , 백엔드 backend



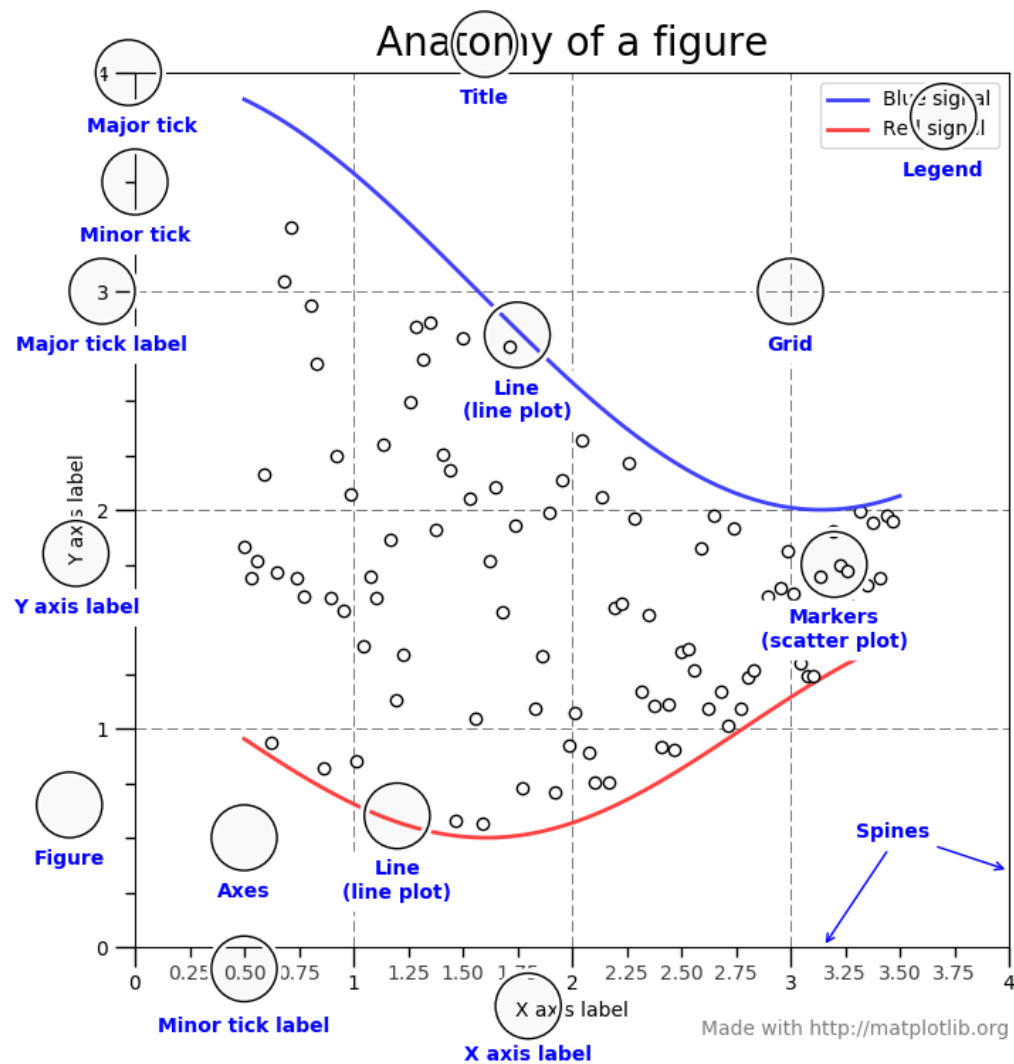


- 스크립팅 레이어
 - Matplotlib가 MATLAB 스크립트처럼 작동하도록 설계된 최상위 계층으로 **절차적 플로팅 수행**
- 아티스트 레이어
 - 스크립팅 레이어에 비해 더 많은 사용자 정의를 수행할 수 있으므로 고급 플롯에 사용
 - 모든 플롯 요소에 대한 최상위 컨테이너인 `Figure` 를 완벽하게 제어하고 미세 조정
 - 렌더러를 사용하여 캔버스에 그리는 **Artis 객체 기반 플로팅 수행**
 - 여러 그림/축을 처리할 때 모든 하위 플롯이 Artis 객체에 할당되기 때문에 현재 활성화된 그림/축을 혼동하지 않음
 - Matplotlib 그림에서 볼 수 있는 모든 것은 Artis 인스턴스
 - 제목, 선, 눈금 레이블, 이미지 등은 모두 개별 Artis
 - Artis 유형
 - 기본 유형: Line2D, Rectangle, Circle, Text
 - 복합 유형: Axis, Tick, Axes, Figure
- 백엔드 레이어
 - PyQt5, ipyml, GTK3, Cocoa, Tk, wxPython과 같은 킷이나 PostScript와 같은 그리기 언어와 통신하여 모든 실제 그리기 작업 처리
 - 대부분의 사용자는 이 계층을 직접 다룰 필요가 없음
 - FigureCanvas: Figure가 렌더링되는 캔버스
 - Renderer: 그리기/렌더링 작업을 처리하는 추상 기본 클래스
 - FigureCanvas에서 그리는 일을 담당
 - Event: 키보드 및 마우스 클릭과 같은 사용자 입력 처리



■ 기본 용어 정리

- Figure: 플롯 전체
- Axes: 플롯이 그려지는 좌표축
 - Figure의 subplot으로 다중 subplot 허용
- Spines: 테두리
- X axis: X 축
- Y axis: Y 축
- Tick : 눈금
 - 주, 보조 눈금으로 나뉨
- Line: 라인 플롯 line plot 의 선
- Markers: 선이나 산포도 scatter 플롯의 점
- Grid: 격자
- Title: 제목
- Label: 각 축이나 눈금 등에 붙이는 텍스트
- Legend : 범례
 - 여러 플롯의 의미를 구분하기 위한 별도 표시





아티스트와 스크립트 레이어 플로팅 비교

■ 아티스트와 스크립트 기반 표준정규분포 플로팅 비교

```
01: from matplotlib.backends.backend_agg import FigureCanvasAgg
02: from matplotlib.figure import Figure
03: import numpy as np
04: from PIL import Image
05:
06: x = np.random.standard_normal(20000)
07:
08: fig = Figure()
09: canvas = FigureCanvasAgg(fig)
10:
11: ax = fig.add_subplot(111)
12:
13: ax.set_title("Normal Distribution")
14: ax.hist(x, 100)
15:
16: canvas.draw()
17: image = np.frombuffer(canvas.tostring_rgb(), dtype=np.uint8)
18: w, h = fig.canvas.get_width_height()
19: im = Image.frombytes("RGB", (w, h), image)
20: im.show()
```

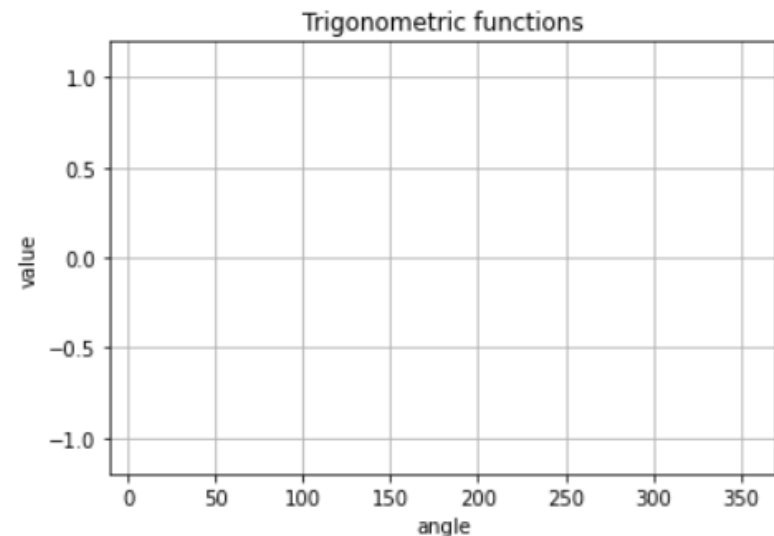
```
01: import matplotlib.pyplot as plt
02: import numpy as np
03:
04: x = np.random.standard_normal(20000)
05:
06: plt.title("Normal Distribution")
07: plt.hist(x, 100)
08:
09: plt.show()
```

플롯 기본

■ 기본 피겨 figure 와 기본 좌표축 axes 에 그리기

- `pyplot.title(name,...)`: 타이틀
 - `loc`: 표시 위치. 'left', 'center', 'right' 중 하나
- `pyplot.xlim(*args, **kwargs)`, `pyplot.ylim(...)`: x, y축 범위
 - `xlime`: left, right, `ylim`: bottom, top
- `pyplot.xlabel(xlabel, loc=None, **kwargs)`, `pyplot.ylabel(...)`: x, y축 이름
 - `loc`: 표시 위치. 'center' 공통. `xlabel`: 'left', 'right'. `ylabel`: 'bottom', 'top'
- `pyplot.grid(visible=None, which='major', axis='both', **kwargs)`: 격자
 - `visible`: 불 타입 표시 유무
 - `axis`: 축 선택. 'x', 'y', 기본값은 None으로 양쪽
 - `color`: '#rrggbb' 또는 색상표 문자열(예: 'red', 'blue', ...) (기본값 자동)
 - `linestyle`: 선 스타일. '-'(기본값), '—', '-.', ':' 중 하나
 - `alpha`: 0.0 ~ 1.0 사이 실수 타입 투명도 (기본값 1)
 - `linewidth`: 정수 타입 선 굵기
- `pyplot.show(*, block=None)`: 모든 그림 표시 및 피겨를 비움
 - `block`: 불 타입으로 True이면 모든 그림이 닫힐 때까지 대기.

```
01: import matplotlib.pyplot as plt
02:
03: plt.title("Trigonometric functions")
04: plt.xlim(-10, 360+10)
05: plt.ylim(-1.2, 1.2)
06: plt.xlabel("angle")
07: plt.ylabel("value")
08: plt.grid(True)
09:
10: plt.show()
```

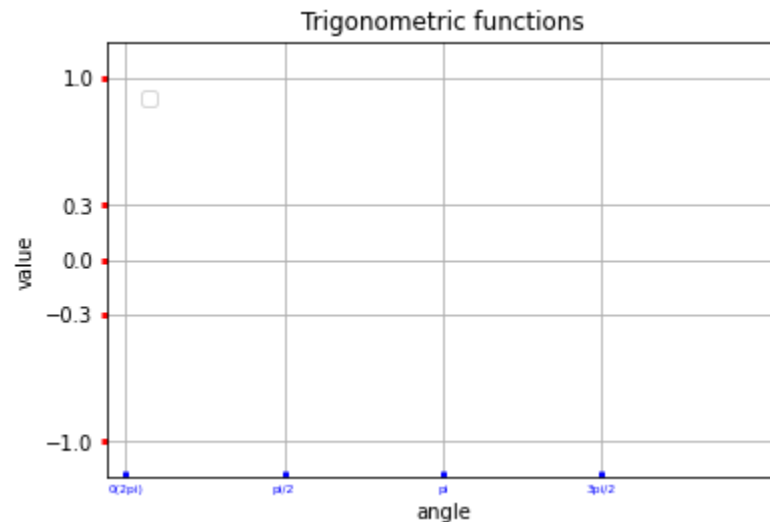




플롯 기본

- `pyplot.xticks(ticks=None, labels=None, **kwargs)`: x축 간격 표시 눈금
 - `ticks`: 리스트 타입 시퀀스. `None`은 표시 안함
 - `labels`: 리스트 타입 시퀀스. `ticks` 값 대신 사용할 이름
- `pyplot.yticks(...)`: y축 간격 표시 눈금
- `pyplot.tick_params(axis='both', **kwargs)`: 축 눈금 세부 설정
 - `axis`: 축 선택. 'x', 'y', 'both' (기본값)
 - `direction`: 축선 기준 눈금 방향으로 'in', 'out' (기본값), 'inout' 중 하나
 - `length`: 정수 타입 눈금 길이. (기본값 3)
 - `width`: 정수 타입 눈금 굵기. (기본값 1)
 - `color`: 눈금 색. (기본값 'black')
 - `labelsize`: 정수 타입 레이블 크기. (기본값은 10)
 - `labelcolor`: 레이블 색. (기본값 'black')
- `pyplot.legend(...)`: 범례
 - `label`: 리스트 타입 시퀀스로 요소의 순서는 플롯팅 순
 - 플롯팅할 플롯이 없으면 표시 안되고 플롯에서 `label`을 사용하면 생략 가능
 - `loc`: 4방향에 대한 0.0 ~ 1.0 실수 타입 튜플(x, y)
 - 생략하면 적당한 곳 자동 선택

```
...
09: plt.xticks([0, 90, 180, 270], ['0(2pi)', 'pi/2', 'pi', '3pi/2'])
10: plt.yticks([-1, -0.3, 0, 0.3, 1])
11: plt.tick_params(axis='x', width=3, direction='in',
12:                 color='blue', labelsize=6, labelcolor='blue')
13: plt.tick_params(axis='y', width=3, color='red')
14: plt.legend(['sine', 'cosine'], loc=(0.05, 0.85))
15:
16: plt.show()
```



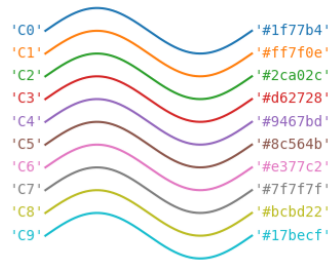
라인 플롯

■ 독립변수 x에 대한 종속변수 y의 변화를 선으로 표현

- `pyplot.plot(*args, **kwargs)`

- 주요 인자

- x: x축 실수 또는 배열 형태 데이터
 - n개의 x축 데이터는 n차원 배열을 사용
- y: y축 실수 또는 배열 형태 데이터
 - 생략하면 x는 x의 인덱스, y는 x의 데이터
- label: 플롯 이름
- color: 색상으로 문자열(예: 'red', 'blue', ...) 또는 RGB 문자열('#rrggbb')
 - 색상을 지정하지 않으면 플롯마다 'C0' ~ 'C9'까지 돌아가며 사용
- linestyle: 선 스타일. '-' (기본값), '—', '-.', ':' 중 하나
- marker: 표시 스타일. '.,ov^<>12348,spP*hH+xDd|_' 중 하나



Cycler 색상

black	bisque	forestgreen	slategrey
dimgray	darkorange	limegreen	lightsteelblue
dimgrey	burlywood	darkgreen	cornflowerblue
gray	antiquewhite	green	royalblue
grey	tan	lime	ghostwhite
darkgray	navajowhite	seagreen	lavender
darkgrey	blanchedalmond	mediumseagreen	midnightblue
silver	papayawhip	springgreen	navy
lightgray	moccasin	mintcream	darkblue
lightgrey	orange	mediumspringgreen	mediumblue
gainsboro	wheat	mediumaquamarine	blue
whitesmoke	oldlace	aquamarine	slateblue
white	floralwhite	turquoise	darkslateblue
snow	darkgoldenrod	lightseagreen	mediumslateblue
rosybrown	goldenrod	mediumturquoise	mediumpurple
lightcoral	cornsilk	azure	rebeccapurple
indianred	gold	lightcyan	blueviolet
brown	lemonchiffon	paleturquoise	indigo
firebrick	khaki	darkslategray	darkorchid
maroon	palegoldenrod	darkslategrey	darkviolet
darkred	darkkhaki	teal	mediumorchid
red	ivory	darkcyan	thistle
mistyrose	beige	aqua	plum
salmon	lightyellow	cyan	violet
tomato	lightgoldenrodyellow	darkturquoise	purple
darksalmon	olive	cadetblue	darkmagenta
coral	yellow	powderblue	fuchsia
orangered	olivedrab	lightblue	magenta
lightsalmon	yellowgreen	deepskyblue	orchid
sienna	darkolivegreen	skyblue	mediumvioletred
seashell	lightyellow	lightskyblue	deeppink
chocolate	chartreuse	steelblue	hotpink
saddlebrown	lawngreen	aliceblue	lavenderblush
sandybrown	honeydew	dodgerblue	palevioletred
peachpuff	darkseagreen	lightslategray	crimson
peru	palegreen	lightslategrey	pink
linen	lightgreen	slategray	lightpink

CSS 색상

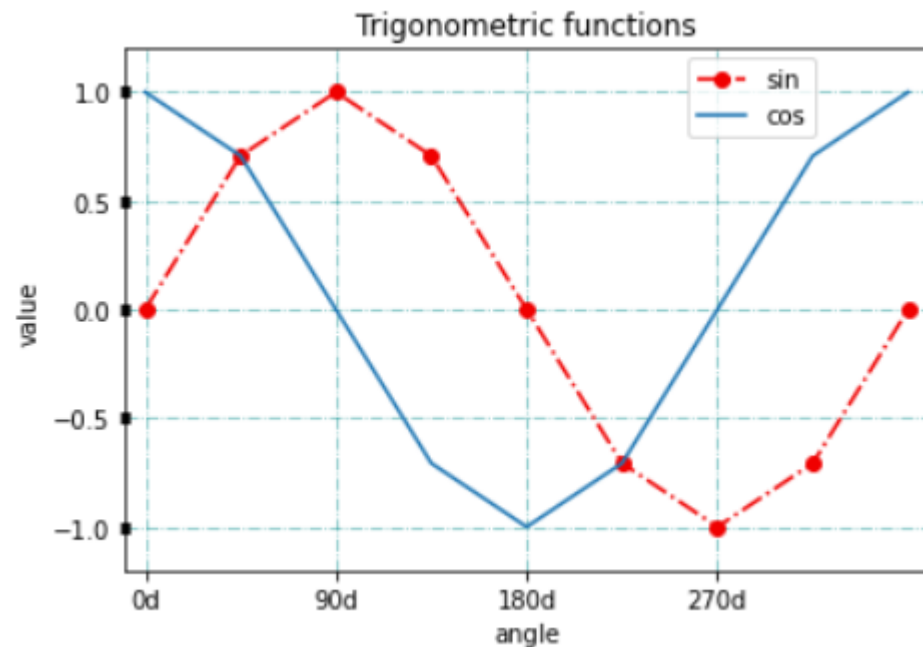


라인 플롯

■ sine, cosine 라인 플로팅

- $0 \sim 2\pi$ 범위 sine, cosine 배열 생성 후 마커(sine 만)와 함께 선 플로팅. step을 줄이면 좀더 부드러운 곡선을 얻을 수 있음

```
01: import matplotlib.pyplot as plt
02: import numpy as np
03:
04: plt.title("Trigonometric functions")
05: plt.xlabel("angle")
06: plt.ylabel("value")
07: plt.grid(True, color='darkcyan', alpha=0.5, linestyle='-.')
08: plt.xlim(-10, 360+10)
09: plt.ylim(-1.2, 1.2)
10:
11: x = np.arange(0, 360+1, 45)
12: sin_y = np.sin(x * np.pi / 180)
13: cos_y = np.cos(x * np.pi / 180)
14:
15: plt.plot(x, sin_y, label='sin', color='#f00000', linestyle='-.', marker='o')
16: plt.plot(x, cos_y, label='cos')
17:
18: plt.legend(loc=(0.7, 0.83))
19: plt.xticks([0, 90, 180, 270], ['0d', '90d', '180d', '270d'])
20: plt.tick_params('y', direction='inout', width=5)
21:
22: plt.show()
```



주가 데이터 얻기

- 전세계 주가는 pandas-datareader 패키지로 야후 파이낸스 사이트(<https://finance.yahoo.com/>)를 통해 얻을 수 있음
 - 설치
 - `sudo pip3 install pandas-datareader`
 - 모듈
 - `from pandas_datareader import data as pdr`
 - 주가 데이터
 - 데이터 구조
 - Date(날짜), High(고가), Low(저가), Open(시가), Close(종가), Volume(거래량), Adj Close(수정종가) 필드로 구성된 테이블 구조
 - 데이터 분석에는 데이터의 연속성을 위해 수정 종가 사용
 - 수정 종가는 분할, 배당, 배분, 신주 발생을 고려해 주식 가격을 조정한 가격
 - 야후 파이낸스는 거래량 데이터의 경우 십자리 이하는 버림
 - 종목 코드로 데이터 얻기
 - `kospi = pdr.DataReader('KS11', 'yahoo')`: 2017년부터 현재까지 코덱스 전체 데이터
 - `sec = pdr.DataReader('005930.KS', 'yahoo', '20200101')`: 삼성전자 2020.01.01 ~ 현재까지
 - `lg = pdr.DataReader('066570.KS', 'yahoo', '20210101', '20211231')`: LG전자 2021년 전체 (2021.01.01 ~ 2021.12.31)



주가 데이터 얻기

- 야후 파이낸스에서 코스피(주식 코드 'KS11')의 주가 데이터를 읽어온 후 ['Adj Close'] 필드를 라인 플롯으로 플롯팅
 - DataReader()는 pandas.core.frame.DataFrame() 객체 반환
 - DataFrame()['Adj Close']은 pandas.core.series.Series() 객체 반환
 - pyplot.plot()는 Series() 객체의 첫 번째 열을 x (Date), 두 번째 열을 y (Adj Close)로 해석

```
01: from pandas_datareader import data as pdr
```

```
02: import matplotlib.pyplot as plt
```

```
03:
```

```
04: kospi = pdr.DataReader('^KS11', 'yahoo')
```

```
05:
```

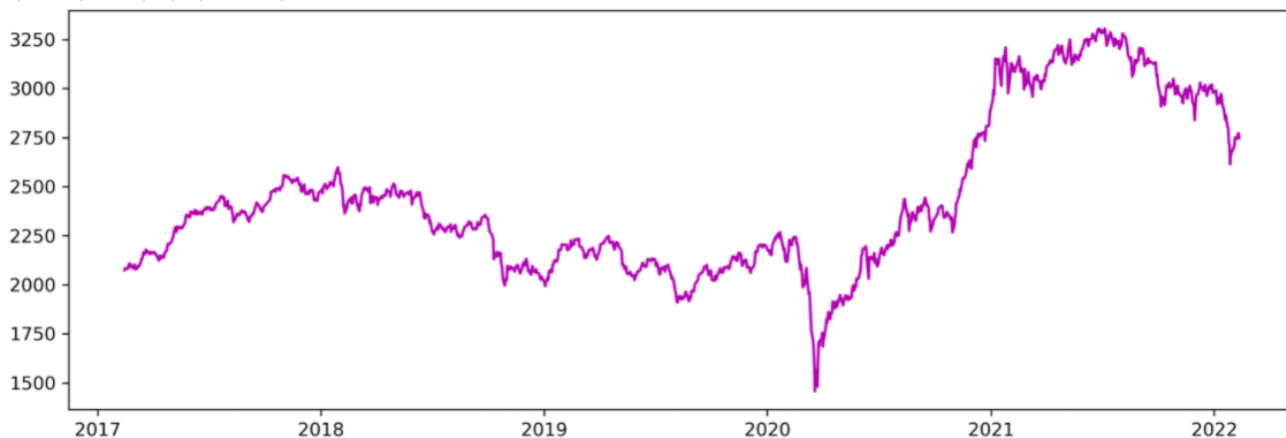
```
06: fig = plt.figure(figsize=(12, 4), dpi=300)
```

```
07: plt.plot(kospi['Adj Close'], label="KOSPI", color='#b000b0')
```

```
08:
```

```
09: plt.show()
```

Date	High	Low	Open	Close	Volume	Adj Close
2017-02-20	2085.590088	2077.129883	2084.159912	2084.389893	297200	2084.389893
2017-02-21	2108.479980	2085.000000	2085.969971	2102.929932	292500	2102.929932
2017-02-22	2108.979980	2101.560059	2106.419922	2106.610107	312200	2106.610107
2017-02-23	2108.989990	2103.110107	2106.149902	2107.629883	430900	2107.629883
2017-02-24	2107.830078	2090.050049	2106.429932	2094.120117	385400	2094.120117
...
2022-02-11	2766.699951	2735.080078	2739.139893	2747.709961	480300	2747.709961
2022-02-14	2724.719971	2688.239990	2715.100098	2704.479980	616000	2704.479980
2022-02-15	2716.449951	2665.469971	2712.449951	2676.540039	588700	2676.540039
2022-02-16	2730.429932	2711.340088	2719.610107	2729.679932	422100	2676.540039
2022-02-17	2770.659912	2711.989990	2735.110107	2744.090088	602261	2729.679932

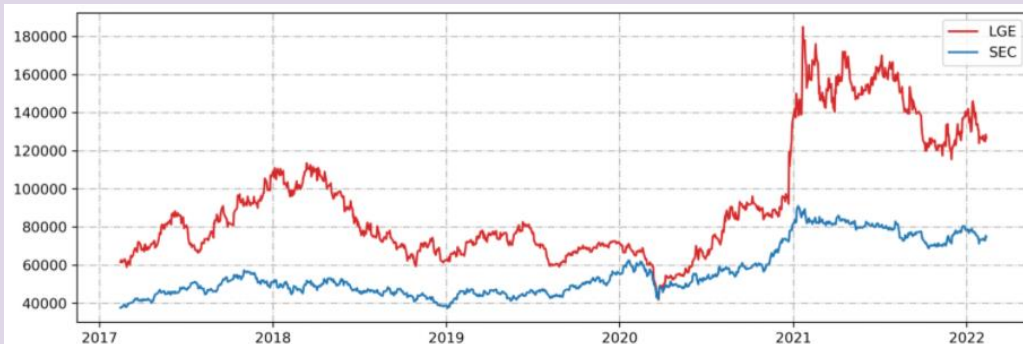




LG전자와 삼성전자 주식 추이를 라인 플롯으로 표시하시오

- 주식 코드 '066570.KS' (LG전자)와 '005930.KR'의 데이터를 읽어온 후 ['Adj Close'] 필드를 라인 플롯으로 플롯팅

```
01: from pandas_datareader import data as pdr
02: import matplotlib.pyplot as plt
03:
04: lge = pdr.DataReader('066570.KS', 'yahoo')
05: sec = pdr.DataReader('005930.KS', 'yahoo')
06:
07: fig = plt.figure(figsize=(12, 4), dpi=300)
08: plt.plot(lge['Close'], label="LGE", color='C3')
09: plt.plot(sec['Close'], label="SEC", color='C0')
10: plt.grid(True, linestyle='-')
11: plt.legend()
12:
13: plt.show()
```



바 플롯

■ 연속적이지 않은 x에 대해 y의 변화를 막대 타입으로 표현

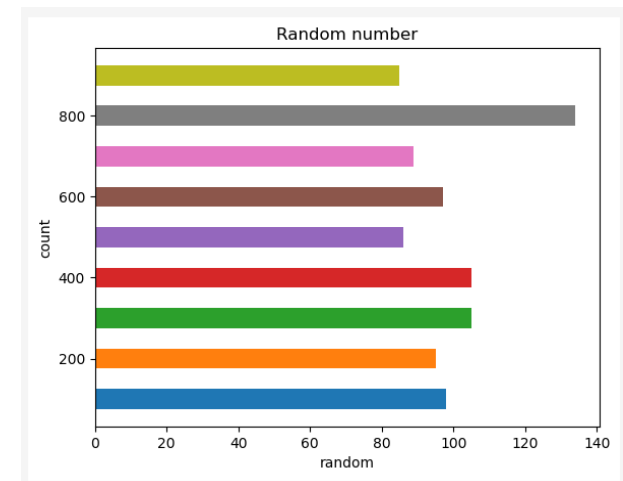
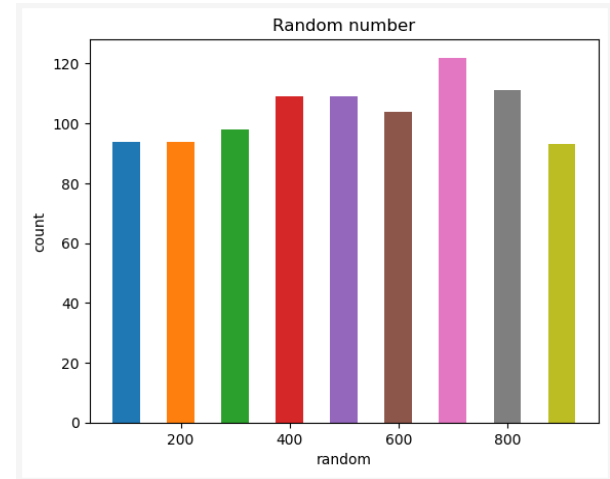
- `pyplot.bar(x, height, width=0.8, bottom=None, *, align='center', **kwargs)`: 세로 막대
 - `x`: 막대의 x 좌표 배열 형태 데이터
 - `height`: 막대 높이의 높이(y 좌표)를 나타내는 배열 형태 데이터
 - `width`: 막대 너비. 기본값은 0.8
 - `Bottom`: 스택 바 타입으로 아래 쪽으로 표시할 막대 높이 배열. (기본값 0)
 - `align`: x 축 눈금 기준 막대 정렬 방법. 'center' (기본값), 'edge'
 - 기타: `color`, `edgecolor`, `linewidth`, `tick_label` 등
- `pyplot.barh(y, with, height=0.8, height=0.8, left=None, align='center', **kwargs)`: 가로 막대
 - `y`: 막대의 y 좌표 배열 형태 데이터
 - `width`: 막대의 폭(x 좌표)을 나타내는 배열 형태 데이터
 - `height`: 막대 높이. 기본값은 0.8

- 난도 발생 빈도를 바 플롯으로 플롯팅
 - $0 \leq n < 1000$ 구간 난수 생성을 100,000회 수행하면서 리스트를 이용해 각 수의 발생 빈도 카운트 (난수값이 리스트의 인덱스)
 - 결과에서 [100::100] 위치의 발생빈도를 바 플롯으로 표시

```

01: import matplotlib.pyplot as plt
02: import numpy as np
03:
04: count = [0] * 1_000
05: for i in range(100_000):
06:     x = np.random.randint(1_000)
07:     count[x] += 1
08:
09: x = np.array([i * 100 for i in range(1, 9+1)])
10: y = np.array(count[100::100])
11:
12: plt.title("Random number")
13: plt.xlabel("random")
14: plt.ylabel("count")
15: plt.bar(x, y, width=50, color=['C0', 'C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8'])
16: plt.show()
17:
18: plt.title("Random number")
19: plt.xlabel("count")
20: plt.ylabel("random")
21: for i in range(len(x)):
22:     plt.barh(x[i], y[i], height=50)
23: plt.show()

```



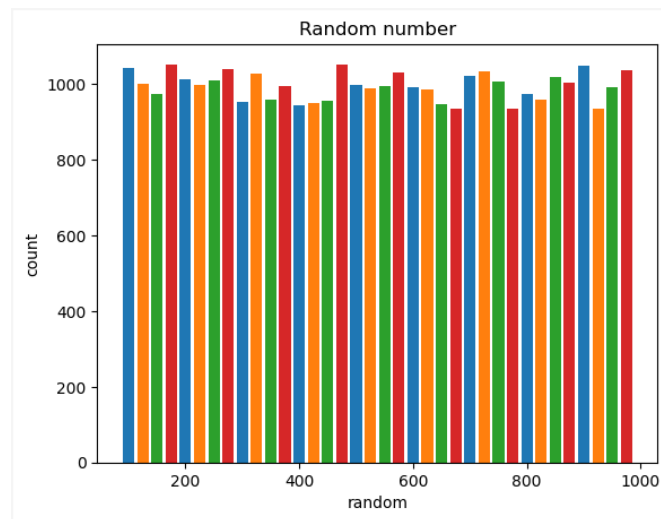


바 플롯

- x가 같고 y가 다른 n개의 바 플롯팅시 중첩 방지
 - 플롯팅 시 일정값 만큼 x 축을 이동해 중첩 방지

```
01: import matplotlib.pyplot as plt
02: import numpy as np
03:
04: def make_rand(n, m):
05:     count = [0] * n
06:     for _ in range(m):
07:         count[np.random.randint(n)] += 1
08:     return count
09:
10: def init_plot():
11:     plt.title("Random number")
12:     plt.xlabel("random")
13:     plt.ylabel("count")
14:
15: def rand_plot(x_offset, n, m, s):
16:     r = make_rand(n, m)
17:     x = np.array([i * 100 for i in range(1, 9+1)])
18:     y = np.array(r[s::s])
19:     plt.bar(x+x_offset, y, width=20)
20:
21: def show_plot():
22:     plt.show()
```

```
23: if __name__ == '__main__':
24:     init_plot()
25:     for x_offset in [0, 25, 50, 75]:
26:         rand_plot(x_offset, 1_000, 100_000, 100)
27:
28:     show_plot()
```

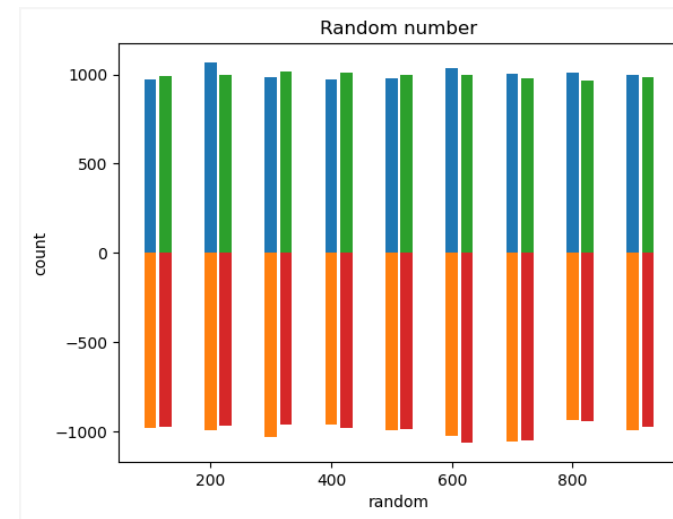




바플롯

- x는 같고 일부 y값을 음수로 바꿔 중첩 방지

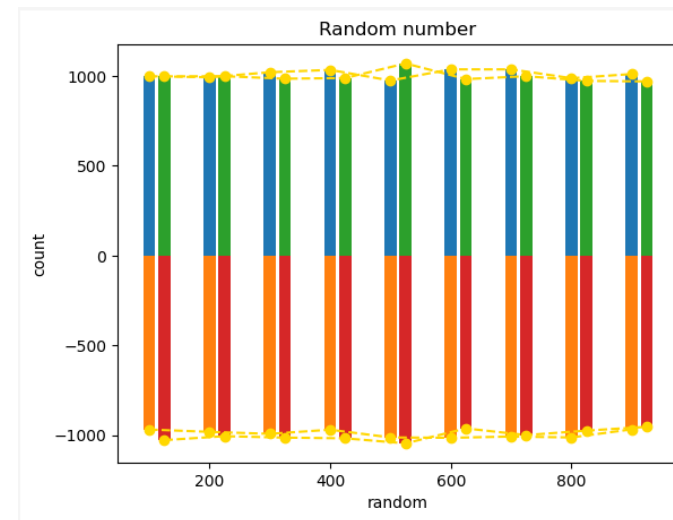
```
...
15: def rand_plot(x_offset, y_offset, n, m, s):
16:     r = make_rand(n, m)
17:     x = np.array([i * 100 for i in range(1, 9+1)])
18:     y = np.array(r[s::s])
19:     plt.bar(x+x_offset, y*y_offset, width=20)
...
23: if __name__ == '__main__':
24:     init_plot()
25:     for x_offset, y_offset in [(0, 1), (0, -1), (25, 1), (25, -1)]:
26:         rand_plot(x_offset, y_offset, 1_000, 100_000, 100)
```



- 라인 플롯으로 추세선 추가

```
...
15: def rand_plot(x_offset, y_offset, n, m, s):
16:     r = make_rand(n, m)
17:     x = np.array([i * 100 for i in range(1, 9+1)])
18:     y = np.array(r[s::s])
19:     plt.bar(x+x_offset, y*y_offset, width=20)
20:     plt.plot(x+x_offset, y*y_offset, color='gold', linestyle='--', marker='o')
...

```





sine, cosine 값을 가로 바 플롯으로 플롯팅하시오

■ 절차를 함수로 구조화해 sine, cosine 값을 가로 바 플롯으로 플롯팅

- data 생성: `make_sincos(a, b)`
 - $0 \leq x < 360$ 사이 x 값 생성
 - `x = arange(0, 360)`
 - x를 라디안으로 바꾼 후 `sin_y`, `cos_y` 값 생성
 - `sin_y, cos_y = sin(x*np.pi/180), cos(y*np.pi/180)`
- 플롯팅: `init_plot()`, `sincos_plot()`, `show_plot()`
 - 타이틀, x, y 레이블, 그리드 추가
 - 생성한 data를 가져와 `barh()`로 `sin_y`, `cos_y` 플롯팅
 - `cos_y`를 플롯팅할 때 정렬을 'edge'로 설정해 `sin_y`와 중첩 최소화
 - x 축 기준 0.8, y 축 기준 0.5 위치에 범례 추가
 - y축 눈금 0, 90, 180, 270에 대해 레이블 '0d', '90d', '180d', '270d' 추가

■ 과연 sine, cosine 값을 바 플롯으로 플롯팅하는 것이 적합한가?

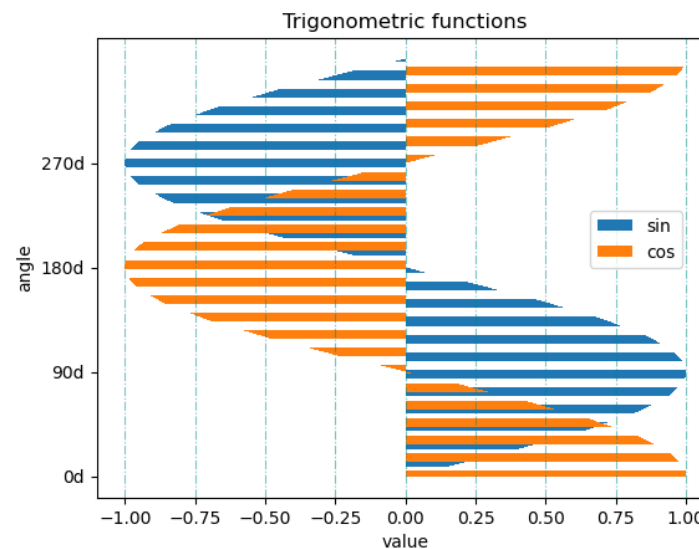


sine, cosine 값을 가로 바 플롯으로 플롯팅하시오

■ sincos_barh_plotting.py

```
01: import matplotlib.pyplot as plt
02: import numpy as np
03:
04: def make_sincos(a, b):
05:     x = np.arange(a, b)
06:     sin_y = np.sin(x * np.pi / 180)
07:     cos_y = np.cos(x * np.pi / 180)
08:     return x, sin_y, cos_y
09:
10: def init_plot():
11:     plt.title("Trigonometric functions")
12:     plt.xlabel("value")
13:     plt.ylabel("angle")
14:     plt.grid(True, axis='x', color='darkcyan', alpha=0.5, linestyle='-'.)
15:
16: def sincos_plot(a, b):
17:     x, sin_y, cos_y = make_sincos(a, b)
18:     plt.barh(x, sin_y, label='sin', height=0.5)
19:     plt.barh(x, cos_y * -1, label='cos', align='edge', height=0.5)
20:
21:     plt.legend(loc=(0.7, 0.83))
22:     plt.yticks([0, 90, 180, 270], ['0d', '90d', '180d', '270d'])
23:
24: def show_plot():
25:     plt.show()
```

```
26:
27: if __name__ == '__main__':
28:     init_plot()
29:     sincos_plot(0, 360)
30:     show_plot()
```



전세계 날씨 데이터 얻기

■ 전세계 날씨는 OpenWeatherMap 권장

- 키 생성을 위해 회원 가입이 필요하며 무료, 유료로 구분됨
 - 회원 가입
 - https://home.openweathermap.org/users/sign_up
 - 무료는 시간당 60회까지 요청 가능. 날씨 업데이트 간격은 2시간 이내
 - 메일 유효성 확인 및 API 키 얻기
 - 가입시 입력한 메일에서 유효성 확인 메일 수신 > Verify your email 체크
 - 메일 유효성이 확인되면 다시 API 키가 포함된 메일이 수신됨
 - ~ 1 시간 정도 지난 후 사용 가능
- 파이썬 패키지 설치
 - `sudo pip3 install pyowm`

How and where will you use our API? X

Hi! We are doing some housekeeping around thousands of our customers. Your impact will be much appreciated. All you need to do is to choose in which exact area you use our services.

Company

* Purpose

Create New Account

We will use information you provided for management and administration purposes, and for keeping you informed by mail, telephone, email and SMS of other products and services from us and our partners. You can proactively manage your preferences or opt-out of communications with us at any time using Privacy Centre. You have the right to access your data held by us or to request your data to be deleted. For full details please see the OpenWeather [Privacy Policy](#).

☒ I am 16 years old and over

☒ I agree with [Privacy Policy](#), [Terms and conditions of sale](#) and [Websites terms and conditions of use](#)


I consent to receive communications from OpenWeather Group of Companies and their partners:


☐ System news (API usage alert, system update, temporary system shutdown, etc)

☐ Product news (change to price, new product features, etc)

☐ Corporate news (our life, the launch of a new service, etc)

☒

로봇이 아닙니다.  개인정보 보호 · 약관



OpenWeather

Dear Customer!

Thank you for choosing [OpenWeatherMap!](#)

Please confirm your email address to help us ensure your account is always protected.

For further technical questions and support, please contact us at info@openweathermap.org


OpenWeather

Dear Customer!

Thank you for subscribing to Free [OpenWeatherMap!](#)

API key:

- Your API key is **9a91e609ccdb5f11d1857d9e5c60de49**
- Within the next couple of hours, it will be activated and ready to use
- You can later create more API keys on your [account page](#)
- Please, always use your API key in each API call

Endpoint:

- Please, use the endpoint api.openweathermap.org for your API calls
- Example of API call:
api.openweathermap.org/data/2.5/weather?q=London,uk&appid=9a91e609ccdb5f11d1857d9e5c60de44



전세계 날씨 데이터 얻기

- 날씨 정보 얻기
 - 수신 메일에서 확인한 API_KEY로 OWM 객체 생성
 - `owm = OWM(API_KEY)`
 - 도시 이름을 인자로 OWM 객체의 `weather_at_place()` 메소드를 호출해 해당 도시의 Observation 객체 반환
 - `observation_seoul = owm.weather_at_place('seoul','kr')`
 - 반환받은 Observation 객체로 `weather_at_place()`와 `get_location()`를 통해 Weather 및 Location 객체 반환
 - `seoul_weather = observation_seoul.weather_at_place():`
 - `Weather.get_temperature():` 온도 딕셔너리 (키: 'temp', 'temp_min', 'temp_max')
 - `Weather.get_humidity():` 습도 정수값
 - `Weather.get_pressure():` 기압 딕셔너리 (키: 'press', 'sea_level')
 - `Weather.get_wind():` 풍향 딕셔너리 (키: 'speed', 'deg')
 - `Weather.get_clouds():` 정수 구름정도
 - `Weather.get_sunrise_time():` 정수 일출시간(unix)
 - `Weather.get_sunset_time():` 정수 일몰시간(unix)
 - `Weather.get_detailed_status():` 문자열 날씨 상태
 - `location_seoul = observation_seoul.get_location()`
 - `get_name():` 문자열 도시이름
 - `get_lon():` 실수 경도
 - `get_lat():` 실수 위도



전세계 날씨 데이터 얻기

- 서울 날씨 및 위치 정보 출력

```
01: from pyowm import OWM
02: from datetime import datetime
03:
04: API_KEY = 'your_api_key'
05: owm = OWM(API_KEY)
06: observation_seoul = owm.weather_at_place('Seoul,KR')
07: seoul_weather = observation_seoul.get_weather()
08: print(seoul_weather)
09:
10: print(seoul_weather.get_temperature('celsius'))
11: print(seoul_weather.get_humidity())
12: print(seoul_weather.get_pressure())
13: print(seoul_weather.get_wind())
14: print(seoul_weather.get_clouds())
15: print(datetime.fromtimestamp(seoul_weather.get_sunrise_time()))
16: print(datetime.fromtimestamp(seoul_weather.get_sunset_time()))
17: print(seoul_weather.get_detailed_status())
18:
19: location_seoul = observation_seoul.get_location()
20: print(location_seoul.get_lon())
21: print(location_seoul.get_lat())
```



서울을 포함해 광역시의 현재 온도를 바 플롯으로 플롯팅하시오

- OpenWeatherMap을 통해 서울, 인천, 대전, 부산, 광주 지역의 현재 온도를 얻어 바 플롯으로 플롯팅
 - data 생성: `make_weather(citys)`
 - 딕셔너리 데이터 준비
 - `{citys: 'seoul':0, 'Incheon':0, 'Daejeon':0, 'pusan':0, 'gwangju':0}`
 - for 루프에서 `keys()`를 분리한 `city`에 `',kr'`를 더해 Observation 객체를 만든 후 Weather 객체에서 온도만 얻어 `citys[city]`에 저장
 - `keys()`와 `values()`를 리스트로 변환한 후 다시 `ndarray`로 변환해 도시 `x`, 온도 `y` 값으로 할당
 - 플롯팅: `init_plot()`, `weather_plot()`, `show_plot()`
 - 타이틀, `x`, `y` 레이블 추가
 - 생성한 data를 가져와 `bar()`로 `x`, `y` 플롯팅
 - 도시별로 색상이 다르게 표시되도록 `color`에 `'C0' ~ 'C4'`까지 할당
 - 막대 위 또는 아래에 온도를 텍스트로 추가할 때 영하 온도는 막대 아래에 표시해야 하므로 `y` 눈금 범위를 각각 0.5씩 확장
 - `pyplot.ylim()`: 현재 `y`축 눈금을 튜플로 반환
 - `Bar()` 객체를 얻어 막대 위 또는 아래에 온도를 텍스트로 표시
 - `bar = pyplot.bar(...)`



서울을 포함해 광역시의 현재 온도를 바 플롯으로 플롯팅하시오

■ make_weather.py

```
01: from pyowm import OWM
02: import numpy as np
03:
04: def make_weather(citys):
05:     API_KEY = '5a91e609ccdb5f11d1857d3e5c60de44'
06:     owm = OWM(API_KEY)
07:
08:     for city in citys.keys():
09:         o = owm.weather_at_place(city + ',kr')
10:         citys[city] = o.get_weather().get_temperature('celsius')['temp']
11:
12:     x = np.array(list(citys.keys()))
13:     y = np.array(list(citys.values()))
14:     return x, y
```



서울을 포함해 광역시의 현재 온도를 바 플롯으로 플롯팅하시오

■ weather_ploting.py

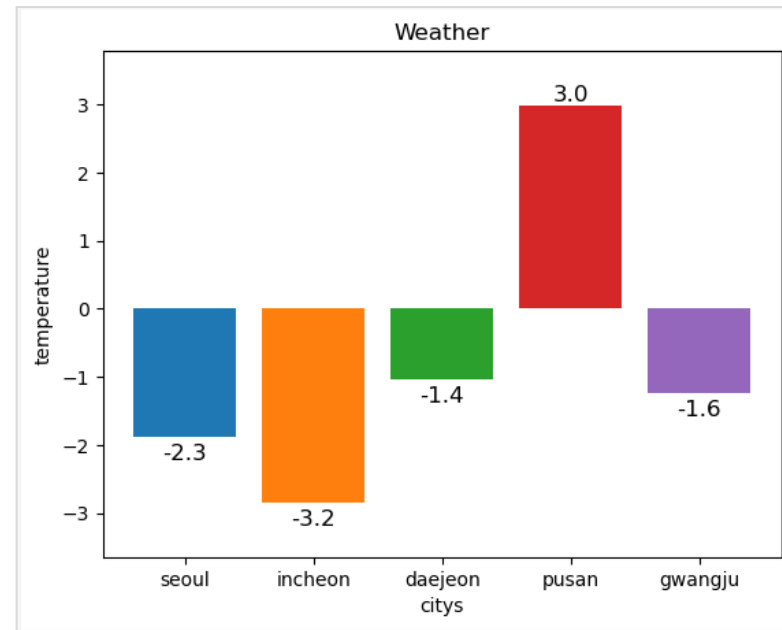
```
01: import matplotlib.pyplot as plt
02: from make_weather import make_weather
03:
04: def init_plot():
05:     plt.title("Weather")
06:     plt.xlabel("citys")
07:     plt.ylabel("temperature")
08:
09: def weather_ploting(citys):
10:     x, y = make_weather(citys)
11:     bar = plt.bar(x, y, color=['C0', 'C1', 'C2', 'C3', 'C4'])
12:
13:     ylim = plt.ylim()           # y에 음수가 있으면 텍스트를 바 플롯에 추가할 때 여유가 있도록 눈금 범위 확장
14:     if (ylim[0] < 0):
15:         plt.ylim((ylim[0] - 0.5, ylim[1] + 0.5))
16:
17:     for rect in bar:           # 바 플롯 위(양수) 또는 아래(음수) 텍스트 추가
18:         height = rect.get_height()
19:         if height < 0:
20:             height -= 0.4
21:         plt.text(rect.get_x() + rect.get_width()/2.0, height, '%.1f' % height, ha='center', va='bottom', size = 12)
22:
23: def show_plot():
24:     plt.show()
```



서울을 포함해 광역시의 현재 온도를 바 플롯으로 플롯팅하시오

■ weather_ploting_main.py

```
01: from weather_ploting import *
02:
03: def main():
04:     citys = {'seoul':0, 'incheon':0, 'daejeon':0, 'pusan':0, 'gwangju':0}
05:     init_plot()
06:     weather_ploting(citys)
07:     show_plot()
08:
09: if __name__ == '__main__':
10:     main()
```



스캐터 플롯

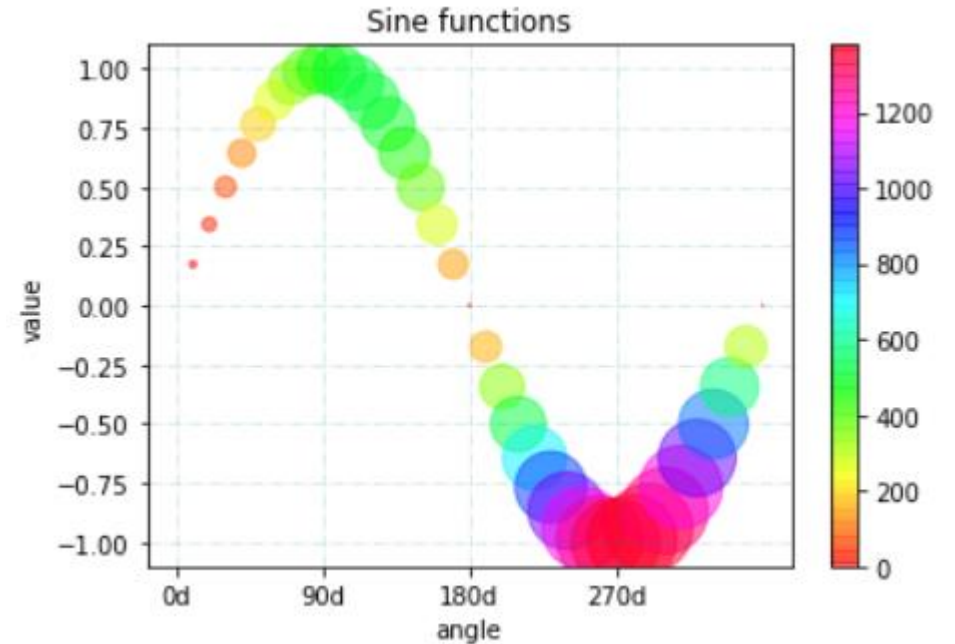
- 서로다른 2개의 독립변수 x_1 와 x_2 의 관계를 산점도로 표현
 - `pyplot.scatter(x, y, s=None, c=None, marker=None, ..., **kwargs)`
 - `x, y`: `x, y` 축 실수 또는 배열 형태 데이터
 - `s`: 실수 또는 배열 형태의 (점 $\times 2$)의 마커 크기. 기본값은 `rcParams['lines.markersize'] $\times 2$`
 - `c`: 배열 형태의 마커 색상
 - `marker`: 마커 스타일. 기본값은 `rcParams['scatter.marker'] == 'o'`
 - `cmap`: `c`가 배열인 경우만 사용하며, 컬러맵 인스턴스 또는 'hsv'와 같은 이름. 기본값은 'viridis'
 - `linewidths`: 실수 또는 배열 형태의 마커의 테두리 선 폭. 기본값은 1.5
 - `edgecolors`: 마커의 테두리 색. 'face'(기본값), 'none' 또는 색상 또는 색상 배열 형태
 - 'face': 면과 동일
 - 'none': 테두리를 그리지 않음
 - `pyplot.colorbar(...)`: 플롯에 색상바 추가



산포도 플롯

- $0 \sim 2\pi$ 범위 sine 값의 크기 변화를 산포도로 표시

```
01: import matplotlib.pyplot as plt
02: import numpy as np
03:
04: plt.title("Trigonometric functions")
05: plt.xlabel("angle")
06: plt.ylabel("value")
07: plt.grid(True, color='darkcyan', alpha=0.2, linestyle='-.')
08:
09: x = np.arange(0, 360+1, 10)
10: sin_y = np.sin(x * np.pi / 180)
11: area = x * np.abs(sin_y) * np.random.randint(10)
12: color = area
13: alpha = 0.5
14:
15: plt.scatter(x, sin_y, s=area, c=color, alpha=alpha, cmap='hsv')
16: plt.colorbar()
17: plt.xticks([0, 90, 180, 270], ['0d', '90d', '180d', '270d'])
18:
19: plt.show()
```



그 밖의 플롯

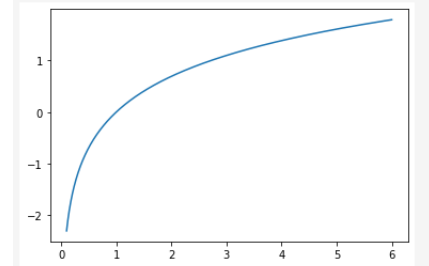
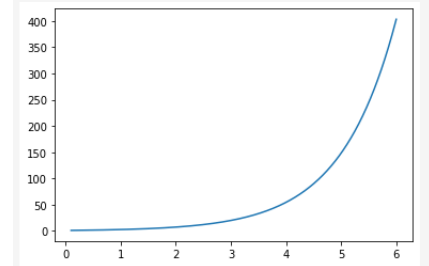
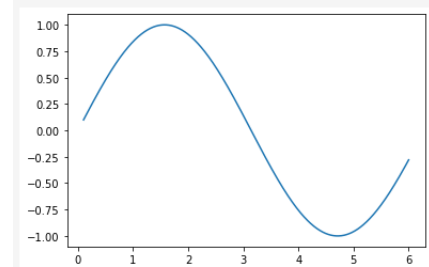
- 히스토그램은 값의 분포를 바 플롯으로 표현
 - 구간별 확률분포나 밀도를 비교하기 좋은 플롯
 - `pyplot.hist(x,..., **kwargs)`
 - `x`: 배열 형태의 데이터
- 파이는 각 값의 비율을 한눈에 비교하기 좋게 원 플롯으로 표현
 - 입력되는 배열 요소는 100개까지 표현 가능
 - `pyplot.pie(x, explode=None, labels=None, colors=None,..., shadow=False,...)`
 - `x`: 1차원 배열 형태로 파이 크기 데이터
 - `explode`: 배열 형태로 각 파이의 반경 비율 지정. 기본값은 `None`
 - `labels`: 배열 형태의 파이 레이블
 - `colors`: 배열 형태의 파이 색상
 - `shadow`: 불 타입의 그림자 표시 유무

서브 플롯팅

■ 여러 좌표축 비교

- `show()`를 호출할 때마다 좌표축이 포함된 현재 피겨를 백엔드 레이어로 전달해 실제 이미지 출력
- 피겨에 포함된 좌표축의 내용을 바꿔가며 `show()` 호출
 - 여러 좌표축 비교로는 적합하지 않음

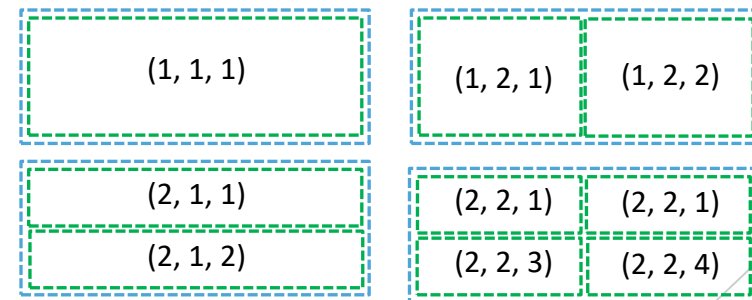
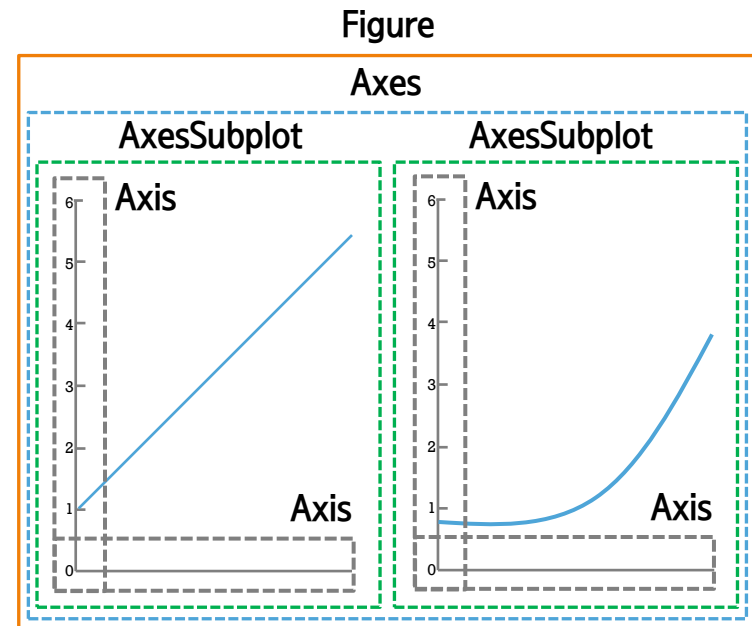
```
01: import matplotlib.pyplot as plt
02: import numpy as np
03:
04: x = np.linspace(0.1, 6.0, 1000)
05: y1 = np.sin(x)
06: y2 = np.exp(x)
07: y3 = np.log(x)
08:
09: plt.plot(x, y1)
10: plt.show()
11: plt.plot(x, y2)
12: plt.show()
13: plt.plot(x, y3)
14: plt.show()
```





서브플롯팅

- Figure와 Axes 객체를 만들지 않고 플롯 함수를 호출하면 스크립팅 레이어는 사용자를 대신해 Figure 및 Axes 객체를 만듦
 - Figure: 모든 플롯 요소에 대한 최상위 컨테이너
 - 피겨 객체를 닫으면 담긴 모든 객체도 함께 제거됨
 - Axes: 데이터 공간에 대한 이미지 영역을 좌표축으로 관리하는 객체
 - 그리드 스타일로 AxesSubplot 배치
 - Axis: 그래프 한계를 설정하고 눈금과 눈금 레이블을 관리하는 숫자 라인과 같은 객체
 - 눈금 위치는 Locator 객체 의해 결정되고 문자열은 포맷터에 의해 형식이 지정됨
- 피겨, 좌표축 객체 참조
 - pyplot.figure() 또는 pyplot.subplots()로 만든 피겨는 지속적으로 참조 유지
 - pyplot.gcf(): 현재 피겨 객체의 참조 반환
 - pyplot.gca(): 현재 좌표축 객체의 참조 반환
 - pyplot.close('all'): 모든 피겨 닫기



gride layout



서브플롯팅

- 피겨에는 원하는 만큼 하위 좌표축을 추가할 수 있으며, 그리드 형태의 레이아웃을 가짐
 - `Figure.add_subplot(*args, **kwargs)`: 지정한 피겨에 좌표축을 추가한 후 Axes의 하위 객체인 `AxesSubplot` 반환
 - 피겨 객체는 `figure.Figure()` 또는 `pyplot.figure()`로 만들
 - `*args`: 좌표축. 기본값은 (1, 1, 1): 기존 좌표축과 다르면 새로 만들고, 같으면 검색
 - 세 개의 정수 (row, column, index): row x column 그리드에서 index로 위치 선택. index는 row 우선
 - 3자리 정수: 서브 플롯이 9개 이하일 때 rci (r=row, c=column, i=index)
 - `projection`: 서브플롯의 투영 유형으로 'aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear' 중 하나. 기본값은 'rectilinear'
 - `sharex, sharey`: x 또는 y 축을 `sharex` 또는 `sharey`와 공유
 - `constrained_layout`: 불 타입으로 True이면 자동으로 좌표축 사이 간격 조정
 - 세부 설정은 `[Figure | pyplot].subplots_adjust(left=None, bottom=None, right=None, top=None, wspace=None, hspace=None)` 메소드나 함수 사용
 - `wspace, hspace`는 실수 타입으로 좌표축 사이 패딩 폭과 높이. `top, bottom, left, right`은 실수 타입으로 좌표축의 위, 아래, 왼쪽, 오른쪽 가장자리 위치
 - `pyplot.subplot(*args, **kwargs)`: 현재 피겨에서 좌표축을 추가하거나 검색한 후 현재 좌표축으로 지정 및 반환
 - 인자는 `add_subplot()`과 같음
 - `pyplot.subplots(*args, **kwargs)`: 새 피겨를 만든 후 Axes 객체를 그리드로 채움
 - `*args`는 행과 열 개수이고, 피겨와 `ndarray` 타입 `AxesSubplot` 객체 시퀀스를 튜플로 반환
 - `add_subplot()`, `subplot()`, `subplots()` 비교
 - `subplot()`은 현재 좌표축 기준
 - `subplots()`와 `add_subplot()`는 반환된 좌표축 기준



서브 플롯팅 방법을 동일한 사례로 비교해 보라

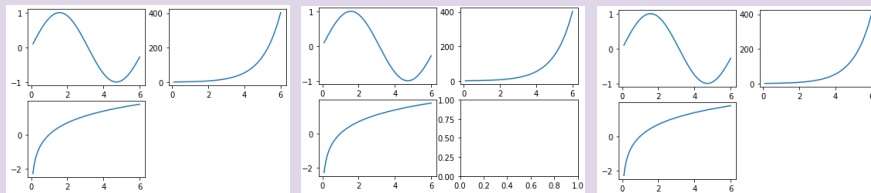
- 0.1 ~ 6.0 구간 100 개의 x 값에 대해 $\sin x$, $\exp x$, $\log x$ 를 계산한 후 `show()`, `subplots()`, `add_subplots()`로 라인 플롯팅

```
01: import matplotlib.pyplot as plt
02: import numpy as np
03:
04: x = np.linspace(0.1, 6.0, 1000)
05: y1 = np.sin(x) ; y2 = np.exp(x) ; y3 = np.log(x)
06:
```

```
07: plt.subplot(221)
08: plt.plot(x, y1)
09: plt.subplot(222)
10: plt.plot(x, y2)
11: plt.subplot(223)
12: plt.plot(x, y3)
13: plt.show()
```

```
07: fig, ((ax1, ax2), (ax3, _)) = plt.subplots(2, 2)
08: ax1.plot(x, y1)
09: ax2.plot(x, y2)
10: ax3.plot(x, y3)
11: plt.show()
```

```
07: fig = plt.figure()
08: ax1 = fig.add_subplot(221)
09: ax2 = fig.add_subplot(222)
10: ax3 = fig.add_subplot(223)
11: ax1.plot(x, y1)
12: ax2.plot(x, y2)
13: ax3.plot(x, y3)
14: plt.show()
```





서브 플롯팅으로 주요 국내 및 미국 주가 추이를 비교해 보라

- 통화 단위 차이로 국내와 미국 주가 추이를 분리해서 플롯팅
 - mystock.py: 주가 차이 얻기
 - 야후 파이낸셜 기준 주식코드
 - LG전자: '066570.KS', 삼성전자: '005930.KS', kospi: '^KS11', nasdaq: '^IXIC', dow30: '^DJI', nasdaq: '^IXIC'
 - mystock_ploting.py: 주가 차이 플롯팅
 - 2x1 그리드를 만든 후 상단에 LG전자, 삼성전자, 코스피를, 하단에 sp500, 다우존스, 나스닥 출력
 - 이미지가 최대한 부드럽게 출력되도록 보간을 'antialiased'로 설정
 - 출력 해상도를 300 dpi 로 설정
 - 피겨 크기를 14 x 14 inch 로 설정
 - 모든 플롯 선 굵기를 1 point 로 설정
 - 모든 좌표축의 눈금 크기를 6 point, 투명도는 0.6으로 설정
 - 모든 좌표축에 격자와 범례 추가
 - 주가 추이를 선으로 플롯팅
 - 피겨를 .png 파일로 저장 및 화면 출력



서브 플롯팅으로 주요 국내 및 미국 주가 추이를 비교해 보라

- mystock.py, mystock_ploting.py 구현

```
01: from pandas_datareader import data as pdr
02:
03: lg = pdr.DataReader('066570.KS', 'yahoo')
04: sec = pdr.DataReader('005930.KS', 'yahoo')
05: kospi = pdr.DataReader('^KS11', 'yahoo')
06:
07: sp500 = pdr.DataReader('^GSPC', 'yahoo')
08: dow30 = pdr.DataReader('^DJI', 'yahoo')
09: nasdaq = pdr.DataReader('^IXIC', 'yahoo')
```

```
01: from matplotlib import rcParams, pyplot as plt
02: import numpy as np
03: from stock import *
04:
05: fig, ((ax_kor), (ax_us)) = plt.subplots(2,1)
06: rcParams['figure.dpi'] = 300 # fig.set_dpi(300)
07: rcParams['figure.figsize'] = (14, 14) # fig.set_size_inches(14, 14)
08: rcParams['lines.linewidth'] = 1
09: rcParams['image.interpolation'] = 'antialiased'
10:
11: for ax in [ax_kor, ax_us]:
12:     ax.tick_params(labelsize=6)
13:     ax.grid(linestyle='-.', alpha=0.4)
14:     ax.legend()
15:
16: ax_kor.plot(lg['Close'], label="LG")
17: ax_kor.plot(sec['Close'], label="Samsung")
18: ax_kor.plot(kospi['Close'], label="KOSPI")
19:
20: ax_us.plot(sp500['Close'], label="SP500")
21: ax_us.plot(dow30['Close'], label="DW30")
22: ax_us.plot(nasdaq['Close'], label="NASDAQ")
23:
24: fig.savefig('mystock.png')
25: fig.show()
```



서브 플롯팅으로 주요 국내 및 미국 주가 추이를 비교해 보라

■ 실행 결과



정리

- 라인 플롯이나 바 플롯은 x 에 대한 y 의 변화를 표현
- 히스토그램 플롯은 하나의 변수 x 에 대한 변화 표현
- 산포도 플롯은 서로다른 2개의 독립변수 x_1 과 x_2 의 관계 표현