Matplotlib 상세 가이드: 데이터 시각화의 모든 것

이 강의 자료는 Python의 대표적인 데이터 시각화 라이브러리인 Matplotlib을 처음 접하는 사용자를 위해 제작되었습니다. Matplotlib의 기본 개념부터 실제 데이터 시각화 프로젝트에 적용할 수 있는 다양한 팁과 모범 사례까지, 상세하고 체계적으로 학습하는 것을 목표로 합니다. 이 자료를 통해 Matplotlib의 기본 사용법을 완벽히 숙달하고, 데이터를 효과적으로 시각화하는 능력을 기를 수 있을 것입니다.

> **참고 자료:**
>
> 이 강의 자료는 **Matplotlib 공식 Quick start guide** 문서를 기반으로 재구성 및 상세화되었습니다.

강의를 시작하기 전에: 기본 설정

Matplotlib을 사용하기 위해서는 먼저 라이브러리를 임포트해야 합니다. 관례적으로 `matplotlib.pyplot`은 `plt`로, `numpy`는 `np`로 줄여서 사용합니다. NumPy는 Matplotlib으로 시각화할 데이터를 생성하고 처리하는 데 필수적인 라이브러리입니다.

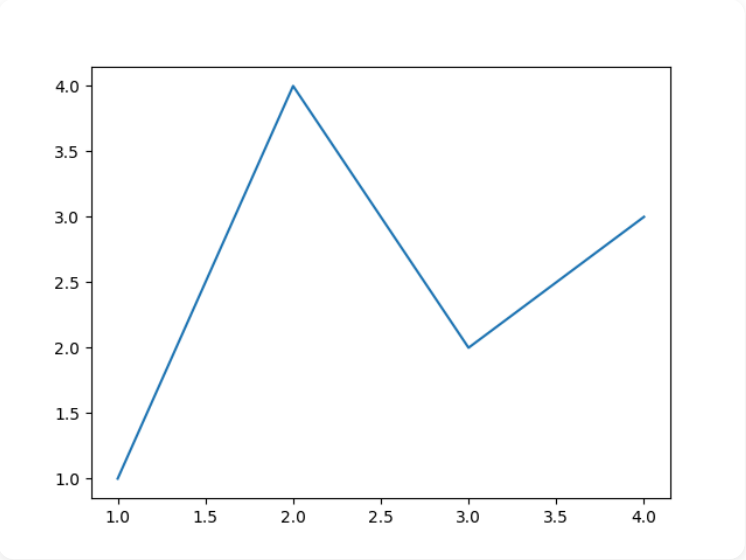
```
import matplotlib.pyplot as plt
import numpy as np
```

1. Matplotlib 시작하기: 간단한 예제

Matplotlib의 가장 기본적인 사용 패턴을 이해하는 것부터 시작하겠습니다. Matplotlib는 **Figure**라는 전체 그림판 위에 하나 이상의 **Axes**(좌표축 영역)를 만들고, 그 위에 데이터를 그리는 방식으로 작동합니다. 가장 간단하게 Figure와 Axes를 생성하는 방법은 `pyplot.subplots` 함수를 사용하는 것입니다. 그 후, `Axes.plot` 메서드를 사용하여 데이터를 그리고, `pyplot.show` 함수로 완성된 그래프를 화면에 표시합니다.

```
# Figure와 그 안에 포함될 단일 Axes를 생성합니다.
fig, ax = plt.subplots()
```

```
# Axes에 데이터를 플로팅합니다. x=[1, 2, 3, 4], y=[1, 4, 2, 3]
ax.plot([1, 2, 3, 4], [1, 4, 2, 3])

# 생성된 Figure를 화면에 보여줍니다.
plt.show()
```

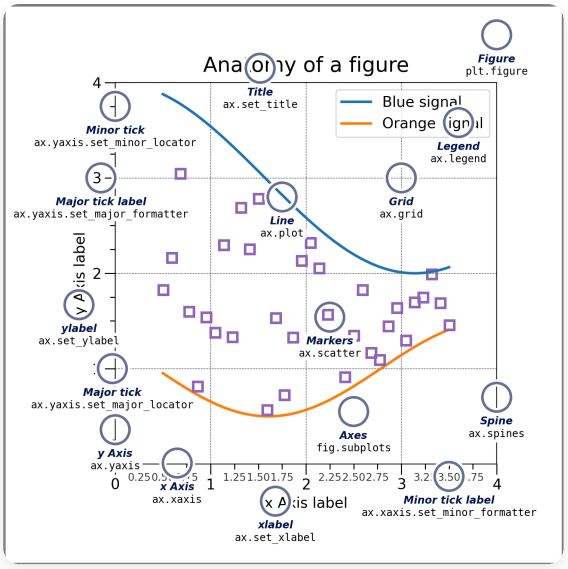


x=[1, 2, 3, 4], y=[1, 4, 2, 3] 데이터를 이용한 간단한 꺾은선 그래프

> **`plt.show()`는 언제 사용해야 할까요?**
>
> Jupyter Notebook이나 JupyterLab과 같은 대화형 환경에서는 코드 셀을 실행하면 생성된 모든 Figure가 자동으로 화면에 표시되므로 `plt.show()`를 생략할 수 있습니다. 하지만 일반적인 Python 스크립트(.py 파일)를 실행할 때는 반드시 `plt.show()`를 호출해야 그래프가 화면에 나타납니다. 스크립트가 끝나기 전에 그래프 창을 띄우고 사용자 입력을 기다리는 역할을 하기 때문입니다.

2. Matplotlib의 핵심 구성 요소: 그림의 해부학

Matplotlib으로 복잡하고 정교한 그래프를 만들기 위해서는 먼저 그래프를 구성하는 핵심 요소들을 이해해야 합니다. 아래 그림은 Matplotlib Figure의 주요 구성 요소와 이를 제어하는 데 사용되는 함수들을 보여줍니다.



Matplotlib Figure의 해부학: Figure, Axes, Axis, Title 등 다양한 구성 요소를 보여줍니다.

이 그림에 나타난 각 구성 요소를 자세히 살펴보겠습니다.

2.1. Figure

Figure는 그래프가 그려지는 전체 창 또는 페이지를 의미합니다. 하나의 Figure는 여러 개의 자식 Axes(서브플롯), 특별한 Artist 그룹(제목, 범례, 컬러바 등), 심지어 중첩된 하위 Figure까지 포함할 수 있습니다. 모든 시각화 요소들을 담는 최상위 컨테이너라고 생각할 수 있습니다.

새로운 Figure를 생성하는 일반적인 방법은 다음과 같습니다.

```
# Axes가 없는 빈 Figure 생성
fig = plt.figure()

# 단일 Axes를 포함하는 Figure 생성
fig, ax = plt.subplots()

# 2x2 그리드 형태의 Axes를 포함하는 Figure 생성
fig, axs = plt.subplots(2, 2)

# 복잡한 레이아웃의 Axes를 포함하는 Figure 생성
# 왼쪽에는 하나의 큰 Axes, 오른쪽에는 위아래로 두 개의 Axes
fig, axs = plt.subplot_mosaic([['left', 'right_top'],
                               ['left', 'right_bottom']])
```

`plt.subplots()`와 `plt.subplot_mosaic()`는 Figure와 그 안의 Axes 객체들을 동시에 생성해주는 편리한 함수입니다. 물론, 빈 Figure를 먼저 만들고 나중에 수동으로 Axes를 추가하는

것도 가능합니다.

2.2. Axes

Axes는 실제 데이터가 그려지는 영역으로, Figure에 속한 Artist입니다. 일반적으로 2개의 **Axis** 객체(3D의 경우 3개)를 포함하며, 이 Axis 객체들이 눈금(tick)과 눈금 레이블(tick label)을 제공하여 데이터의 스케일을 나타냅니다. **'Axes'(축들, 복수형)와 'Axis'(축, 단수형)의 차이점**에 유의해야 합니다. Axes는 전체 좌표평면 영역을, Axis는 x축 또는 y축 자체를 의미합니다.

각 Axes는 제목(`set_title()`), x축 레이블(`set_xlabel()`), y축 레이블(`set_ylabel()`)을 가질 수 있습니다. 대부분의 그래프 설정(데이터 추가, 축 스케일 및 범위 제어, 레이블 추가 등)은 이 Axes 객체의 메서드를 통해 이루어집니다. 따라서 Axes는 Matplotlib 시각화의 가장 핵심적인 인터페이스입니다.

2.3. Axis

Axis 객체는 데이터의 스케일과 한계를 설정하고, 축 위의 눈금(tick)과 눈금 레이블(ticklabel)을 생성하는 역할을 합니다. 눈금의 위치는 **Locator** 객체에 의해 결정되고, 눈금 레이블의 문자열 형식은 **Formatter** 객체에 의해 결정됩니다. 적절한 Locator와 Formatter를 조합하면 눈금의 위치와 레이블을 매우 정밀하게 제어할 수 있습니다.

2.4. Artist

기본적으로 Figure 위에 보이는 모든 것은 **Artist**입니다. 여기에는 `Figure`, `Axes`, `Axis` 객체뿐만 아니라, `Text` 객체, `Line2D` 객체(선 그래프), `collections` 객체(산점도), `Patch` 객체(막대 그래프, 도형) 등이 모두 포함됩니다. Figure가 렌더링될 때, 모든 Artist들이 캔버스(canvas)에 그려집니다. 대부분의 Artist는 특정 Axes에 종속되어 있으며, 여러 Axes에서 공유되거나 다른 Axes로 이동할 수 없습니다.

3. 플로팅 함수에 대한 입력 유형

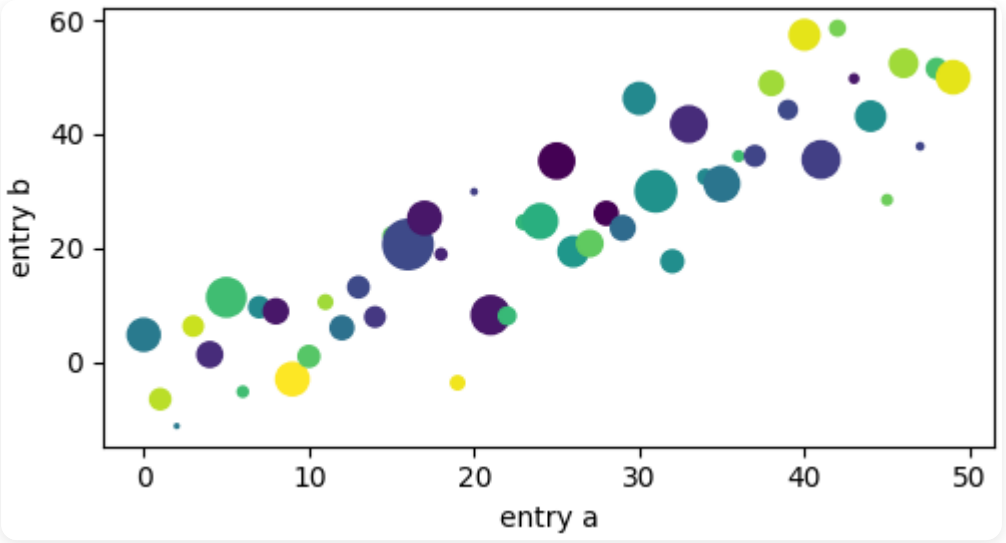
Matplotlib의 플로팅 함수들은 기본적으로 `numpy.array` 또는 `numpy.ma.masked_array`를 입력으로 기대합니다. 또한 `numpy.asarray` 함수로 변환될 수 있는 객체들도 대부분 지원됩니다. 하지만 `pandas`의 데이터 객체나 `numpy.matrix`와 같은 배열과 유사한(array-like) 클래스들은 의도대로 작동하지 않을 수 있습니다. 일반적인 관례는 플로팅하기 전에 이러한 객체들을 `numpy.array`로 변환하는 것입니다.

```
# numpy.matrix를 numpy.array로 변환하는 예시
b = np.matrix([[1, 2], [3, 4]])
b_asarray = np.asarray(b)
```

`data` 키워드 인자 활용하기

대부분의 메서드는 `dict`, 구조화된 NumPy 배열(structured numpy array), 또는 `pandas.DataFrame`과 같이 문자열로 인덱싱이 가능한 객체를 파싱할 수 있습니다. Matplotlib는 `data` 키워드 인자를 제공하여, x와 y 변수에 해당하는 문자열을 전달하여 그래프를 생성할 수 있게 해줍니다. 이는 코드를 더 명확하고 가독성 있게 만들어 줍니다.

```
# 재현성을 위한 랜덤 시드 설정
np.random.seed(19680801)

# 딕셔너리 형태의 데이터 생성
data = {'a': np.arange(50),
        'c': np.random.randint(0, 50, 50),
        'd': np.random.randn(50)}
data['b'] = data['a'] + 10 * np.random.randn(50)
data['d'] = np.abs(data['d']) * 100

fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')

# data 인자를 사용하여 산점도 그리기
# x='a', y='b', color='c', size='d'
ax.scatter('a', 'b', c='c', s='d', data=data)
ax.set_xlabel('entry a')
ax.set_ylabel('entry b')

plt.show()
```



`data` 인자를 사용하여 딕셔너리의 키를 변수로 지정한 산점도. 점의 색상과 크기가 다른 변수에 매핑되어 있습니다.

4. 코딩 스타일: 객체 지향(OO) 스타일과 Pyplot 스타일

Matplotlib을 사용하는 방법에는 크게 두 가지가 있습니다. 이 두 가지 스타일의 차이점을 이해하고 상황에 맞게 사용하는 것이 중요합니다.

- **객체 지향(Object-Oriented, OO) 스타일:** `Figure`와 `Axes` 객체를 명시적으로 생성하고, 이 객체들의 메서드를 호출하여 그래프를 그립니다.
- **Pyplot 스타일:** `pyplot`이 암시적으로 Figure와 Axes를 생성하고 관리하도록 의존하며, `pyplot` 함수를 사용하여 그래프를 그립니다.

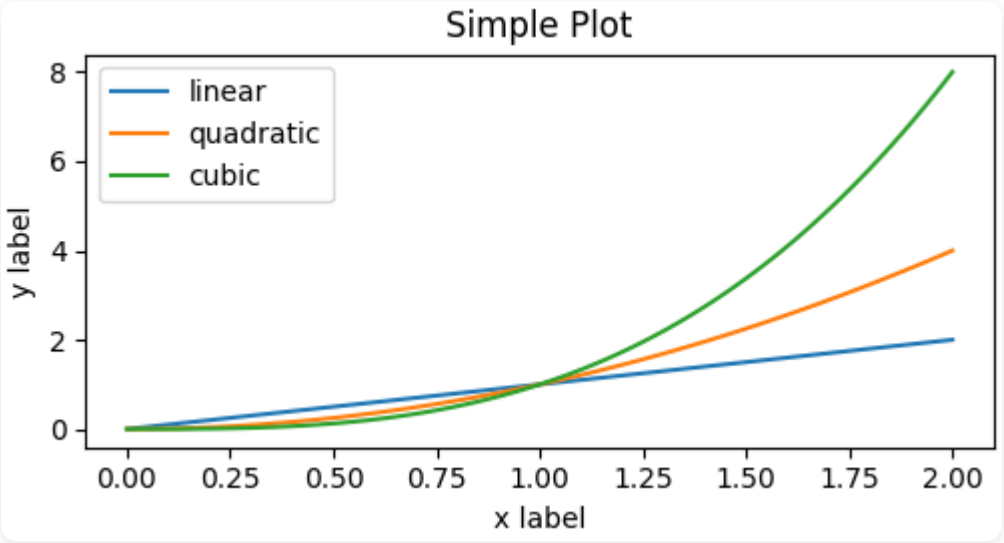
일반적으로 복잡한 그래프를 그리거나, 더 큰 프로젝트의 일부로 재사용될 함수나 스크립트를 작성할 때는 **객체 지향 스타일을 사용하는 것이 좋습니다.** 반면, 빠르고 간단한 대화형 작업에는 Pyplot 스타일이 매우 편리할 수 있습니다.

4.1. 객체 지향 (OO) 스타일

이 스타일에서는 `plt.subplots()`와 같은 함수로 `fig`와 `ax` 객체를 직접 받아온 후, 모든 플로팅과 설정을 `ax` 객체의 메서드(예: `ax.plot()`, `ax.set_xlabel()`)를 통해 수행합니다. 하나의 Figure에 여러 개의 서브플롯이 있을 때 각 서브플롯을 명확하게 제어할 수 있어 코드가 더 명시적이고 구조화됩니다.

```
x = np.linspace(0, 2, 100)  # 샘플 데이터

# OO 스타일에서도 Figure 생성을 위해 pyplot을 사용합니다.
```

```
fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')

# Axes 객체의 메서드를 사용하여 데이터를 플로팅합니다.
ax.plot(x, x, label='linear')
ax.plot(x, x**2, label='quadratic')
ax.plot(x, x**3, label='cubic')

# Axes 객체의 메서드를 사용하여 레이블과 제목을 추가합니다.
ax.set_xlabel('x label')
ax.set_ylabel('y label')
ax.set_title("Simple Plot")
ax.legend()  # 범례 추가

plt.show()
```



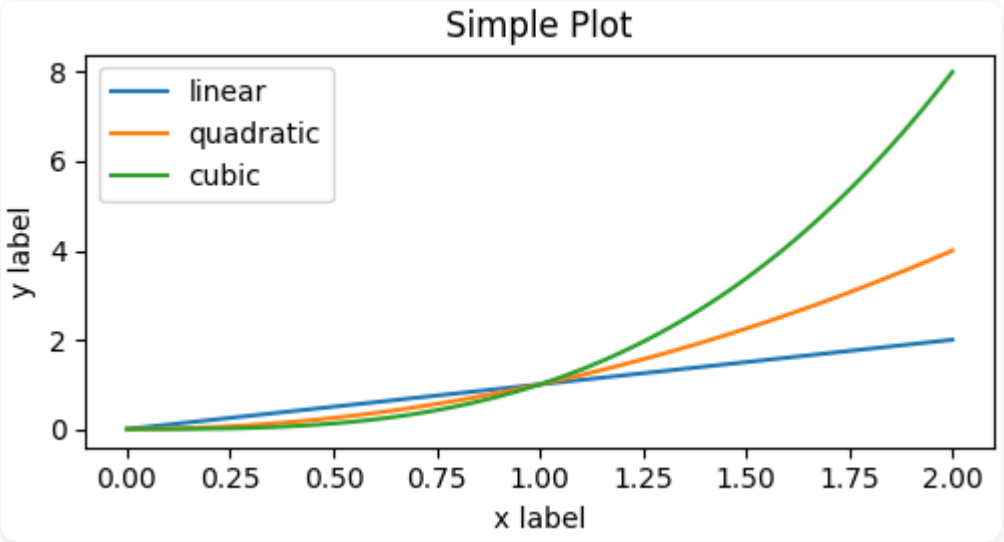
객체 지향(OO) 스타일로 생성된 세 개의 함수 그래프.

4.2. Pyplot 스타일

이 스타일은 Matplotlib이 "현재" Figure와 "현재" Axes를 내부적으로 추적하도록 합니다. `plt.plot()`과 같은 `pyplot` 함수를 호출하면, Matplotlib은 자동으로 현재 Axes에 그래프를 그립니다. 코드가 더 간결해 보이지만, 여러 그래프나 복잡한 레이아웃을 다룰 때는 어떤 Axes에 작업이 수행되는지 혼란스러울 수 있습니다.

```
x = np.linspace(0, 2, 100)  # 샘플 데이터

plt.figure(figsize=(5, 2.7), layout='constrained')
```

```
# pyplot 함수를 사용하여 (암시적인) Axes에 데이터를 플로팅합니다.
plt.plot(x, x, label='linear')
plt.plot(x, x**2, label='quadratic')
plt.plot(x, x**3, label='cubic')

# pyplot 함수를 사용하여 레이블과 제목을 추가합니다.
plt.xlabel('x label')
plt.ylabel('y label')
plt.title("Simple Plot")
plt.legend()

plt.show()
```



Pyplot 스타일로 생성된 세 개의 함수 그래프. 결과는 OO 스타일과 동일합니다.

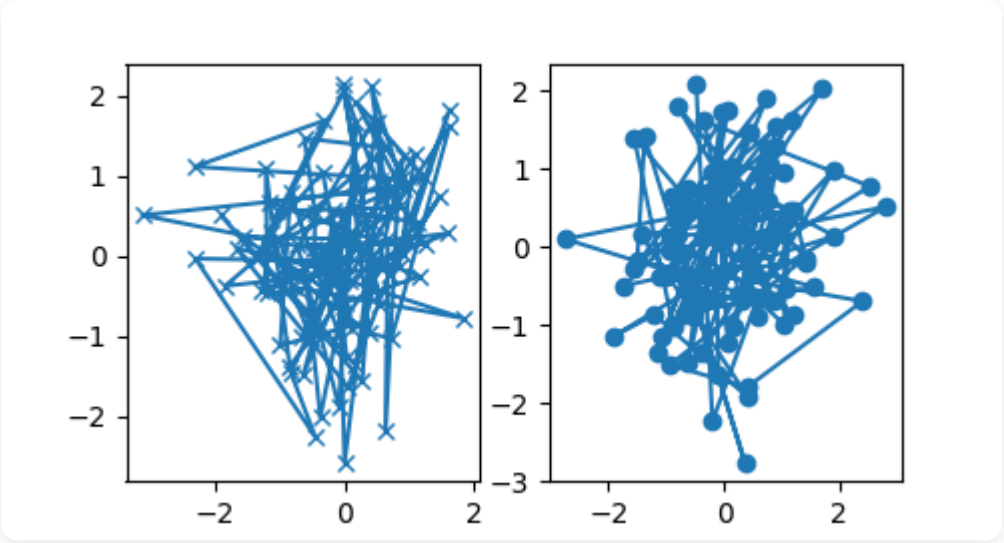
> **주의: `pylab` 인터페이스 사용 금지**
>
> 오래된 예제 코드에서 `from pylab import *`와 같은 구문을 볼 수 있습니다. 이 방식은 Matplotlib과 NumPy의 많은 함수를 전역 네임스페이스로 가져와 이름 충돌을 일으킬 수 있으므로
> **강력히 비권장됩니다.**

4.3. 헬퍼 함수 만들기

동일한 종류의 그래프를 다른 데이터셋으로 반복해서 그려야 할 경우, 플로팅 로직을 헬퍼 함수로 캡슐화하는 것이 좋습니다. 이때 함수가 `Axes` 객체를 첫 번째 인자로 받도록 설계하면, 어

떤 서브플롯에든 이 함수를 재사용하여 그래프를 그릴 수 있습니다. 이는 객체 지향 스타일의 큰 장점 중 하나입니다.

```
def my_plotter(ax, data1, data2, param_dict):
    """
    그래프를 그리는 헬퍼 함수.
    """
    out = ax.plot(data1, data2, **param_dict)
    return out

# 4개의 랜덤 데이터셋 생성
data1, data2, data3, data4 = np.random.randn(4, 100)

# 1x2 서브플롯 생성
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(5, 2.7))

# 첫 번째 서브플롯에 헬퍼 함수 사용
my_plotter(ax1, data1, data2, {'marker': 'x'})

# 두 번째 서브플롯에 헬퍼 함수 사용
my_plotter(ax2, data3, data4, {'marker': 'o'})

plt.show()
```



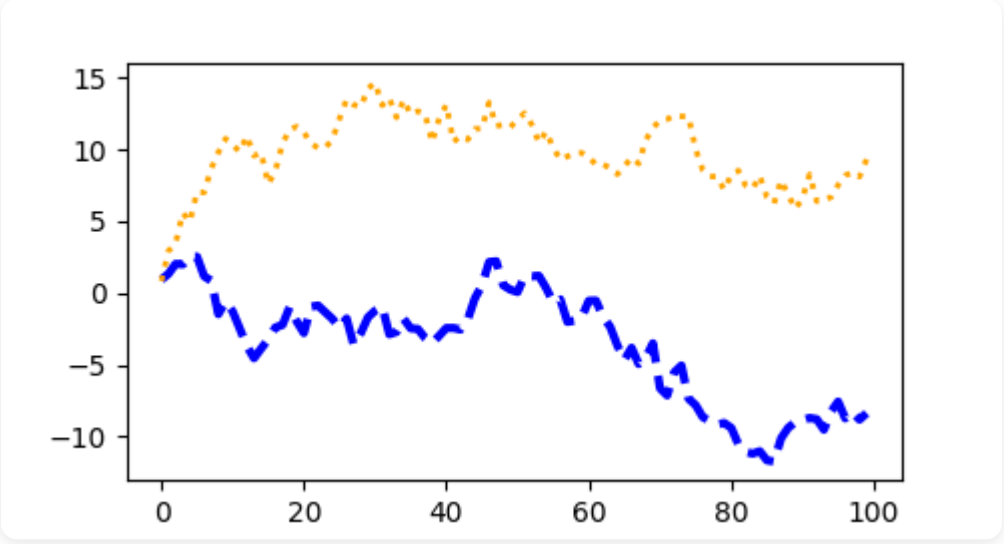
헬퍼 함수를 재사용하여 두 개의 다른 서브플롯에 그래프를 그린 예시.

5. 그래프 스타일링: Artist 속성 꾸미기

대부분의 플로팅 메서드는 생성되는 Artist의 스타일을 지정할 수 있는 옵션을 제공합니다. 이러한 스타일은 플로팅 메서드를 호출할 때 키워드 인자로 전달하거나, 생성된 Artist 객체의 "setter" 메서드를 통해 나중에 설정할 수 있습니다.

아래 예제에서는 `plot`으로 생성된 Artist의 색상, 선 너비, 선 스타일을 직접 설정하고, 두 번째 선의 스타일은 나중에 `set_linestyle` 메서드를 사용하여 변경합니다.

```
fig, ax = plt.subplots(figsize=(5, 2.7))
x = np.arange(len(data1))

# 플로팅 시점에 스타일 지정
ax.plot(x, np.cumsum(data1), color='blue', linewidth=3, linestyle='--')

# Artist 객체를 받아온 후, setter 메서드로 스타일 변경
l, = ax.plot(x, np.cumsum(data2), color='orange', linewidth=2)
l.set_linestyle(':')

plt.show()
```

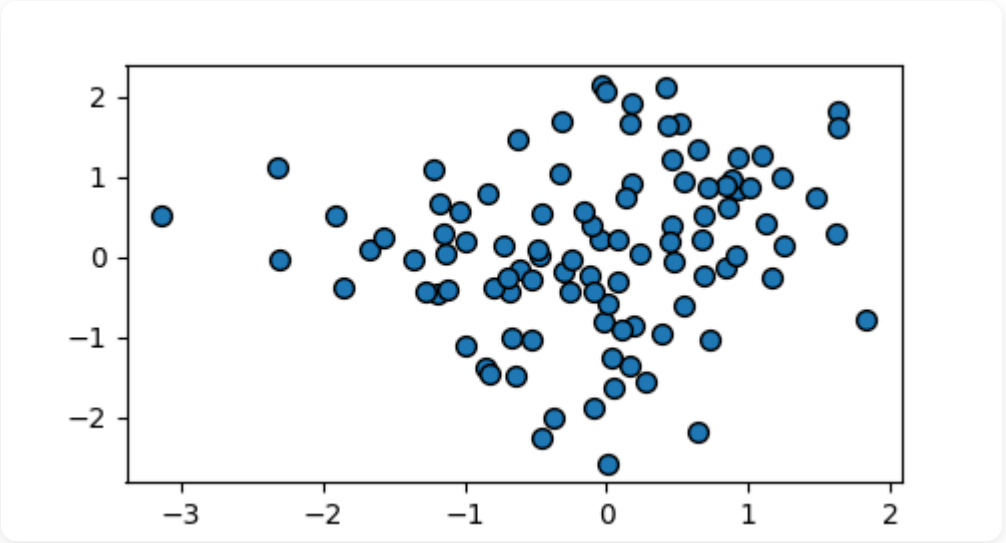


다양한 선 스타일(색상, 너비, 종류)이 적용된 두 개의 누적 합계 그래프.

5.1. 색상 (Colors)

Matplotlib는 대부분의 Artist에 적용할 수 있는 매우 유연한 색상 배열을 지원합니다. (자세한 내용은 **색상 정의 가이드** 참조) `scatter` 플롯과 같이 일부 Artist는 여러 색상을 가질 수 있습니다. 예를 들어, 마커의 테두리 색과 내부 색을 다르게 지정할 수 있습니다.

```
fig, ax = plt.subplots(figsize=(5, 2.7))
ax.scatter(data1, data2, s=50, facecolor='C0', edgecolor='k')
```



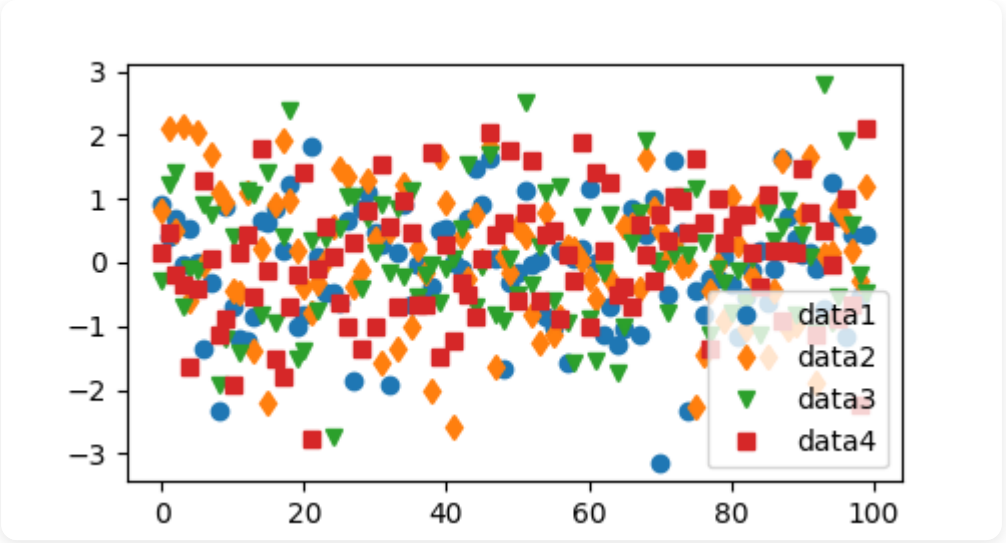
내부 색상(`facecolor='C0'`, 파란색 계열)과 테두리 색상(`edgecolor='k'`, 검은색)이 다르게 지정된 산점도.

5.2. 선 너비, 선 스타일, 마커 크기

선 너비(Linewidths)는 일반적으로 타이포그래피 포인트(1 pt = 1/72 인치) 단위이며, 선이 있는 Artist에 사용할 수 있습니다. 마찬가지로, 선에는 다양한 **선 스타일(linestyles)**을 적용할 수 있습니다. (예시: **선 스타일 예제**)

마커 크기(Markersizes)는 사용되는 메서드에 따라 다릅니다. `plot` 메서드는 마커 크기를 포인트 단위로 지정하며, 일반적으로 마커의 직경이나 너비를 의미합니다. `scatter` 메서드는 마커 크기를 마커의 시각적 면적에 비례하도록 지정합니다. 문자열 코드로 다양한 마커 스타일을 사용할 수 있으며, 사용자가 직접 `MarkerStyle`을 정의할 수도 있습니다.

```
fig, ax = plt.subplots(figsize=(5, 2.7))
ax.plot(data1, 'o', label='data1')  # 원형 마커
ax.plot(data2, 'd', label='data2')  # 다이아몬드 마커
ax.plot(data3, 'v', label='data3')  # 역삼각형 마커
ax.plot(data4, 's', label='data4')  # 사각형 마커
ax.legend()

plt.show()
```



네 개의 데이터셋을 각각 다른 마커 스타일(원, 다이아몬드, 역삼각형, 사각형)로 표현한 그래프.

6. 그래프에 레이블 추가하기

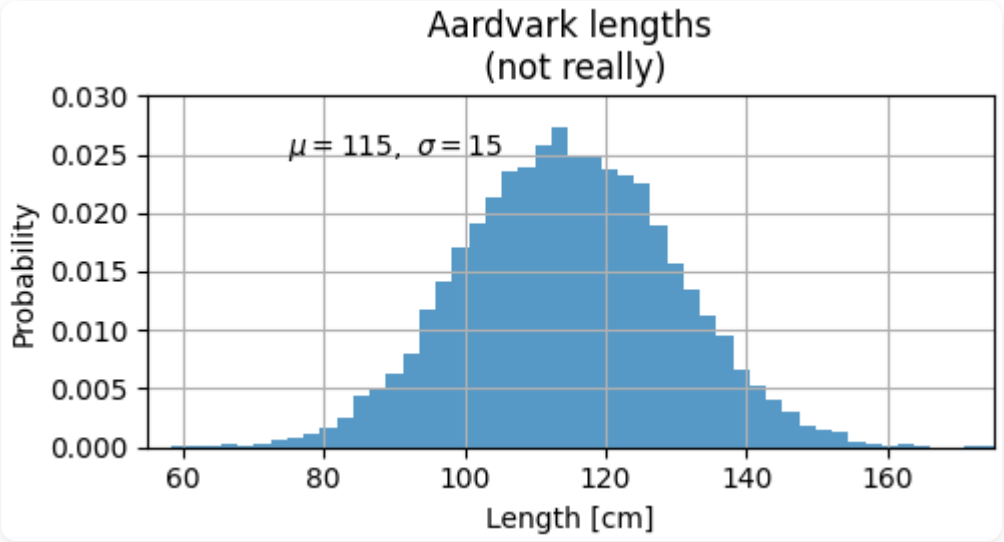
데이터를 시각화할 때, 그래프가 무엇을 의미하는지 명확하게 전달하는 것이 매우 중요합니다. 이를 위해 제목, 축 레이블, 텍스트, 주석, 범례 등을 추가할 수 있습니다.

6.1. 축 레이블과 텍스트

`set_xlabel()`, `set_ylabel()`, `set_title()`은 각각 지정된 위치에 텍스트를 추가하는 데 사용됩니다. 또한 `text()` 메서드를 사용하면 그래프의 특정 좌표에 직접 텍스트를 추가할 수 있습니다.

```
# 정규분포를 따르는 데이터 생성
mu, sigma = 115, 15
x = mu + sigma * np.random.randn(10000)

fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')

# 데이터의 히스토그램 그리기
ax.hist(x, 50, density=True, facecolor='C0', alpha=0.75)

ax.set_xlabel('Length [cm]')
ax.set_ylabel('Probability')
ax.set_title('Aardvark lengths\n(not really)')
ax.text(75, .025, r'$\mu=115,\ \sigma=15$') # 수학식 텍스트 추가
ax.axis([55, 175, 0, 0.03]) # 축 범위 설정
ax.grid(True) # 그리드 표시
```

```
plt.show()
```



제목, 축 레이블, 그리드 및 수학식이 포함된 텍스트가 추가된 히스토그램.

6.2. 텍스트에 수학식 사용하기

Matplotlib는 모든 텍스트 표현식에서 TeX 수식 표현을 지원합니다. 예를 들어, 제목에 $\sigma_i=15$ 라는 수식을 쓰려면, 달러 기호($)로 둘러싸인 TeX 표현식을 작성하면 됩니다.

```
ax.set_title(r'$\sigma_i=15$')
```

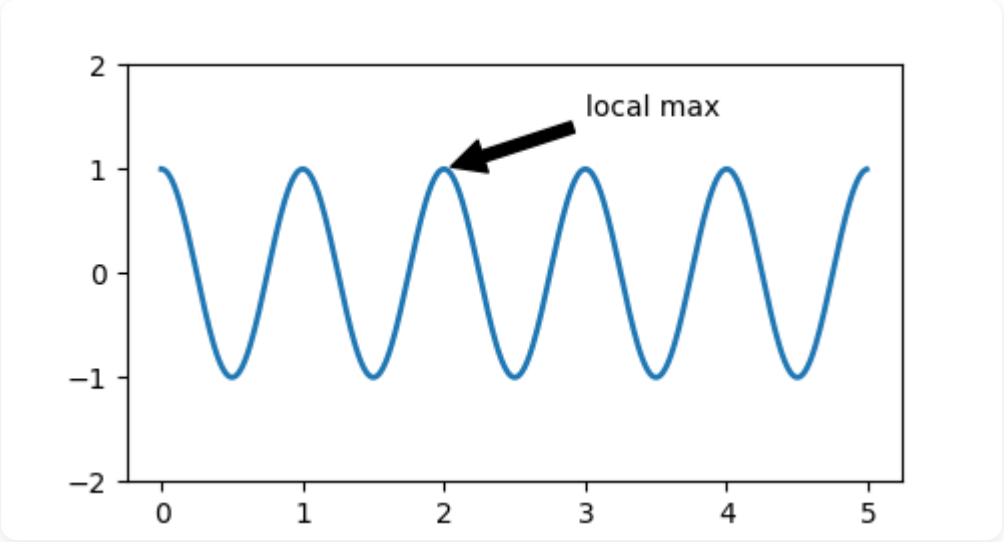
여기서 제목 문자열 앞의 `r`은 해당 문자열이 'raw string'임을 의미하며, 백슬래시를 Python 이스케이프 문자로 처리하지 않도록 합니다. Matplotlib는 내장된 TeX 표현식 파서와 레이아웃 엔진을 가지고 있으며, 자체 수학 폰트를 제공합니다. (자세한 내용은 **수학식 작성 가이드** 참조)

6.3. 주석 (Annotations)

그래프 위의 특정 지점을 강조하고 싶을 때 주석을 사용할 수 있습니다. 종종 화살표를 사용하여 텍스트(`xytext`)와 특정 데이터 지점(`xy`)을 연결합니다.

```
fig, ax = plt.subplots(figsize=(5, 2.7))

t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2 * np.pi * t)
line, = ax.plot(t, s, lw=2)
```

```
# 주석 추가
ax.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
            arrowprops=dict(facecolor='black', shrink=0.05))

ax.set_ylim(-2, 2)
plt.show()
```



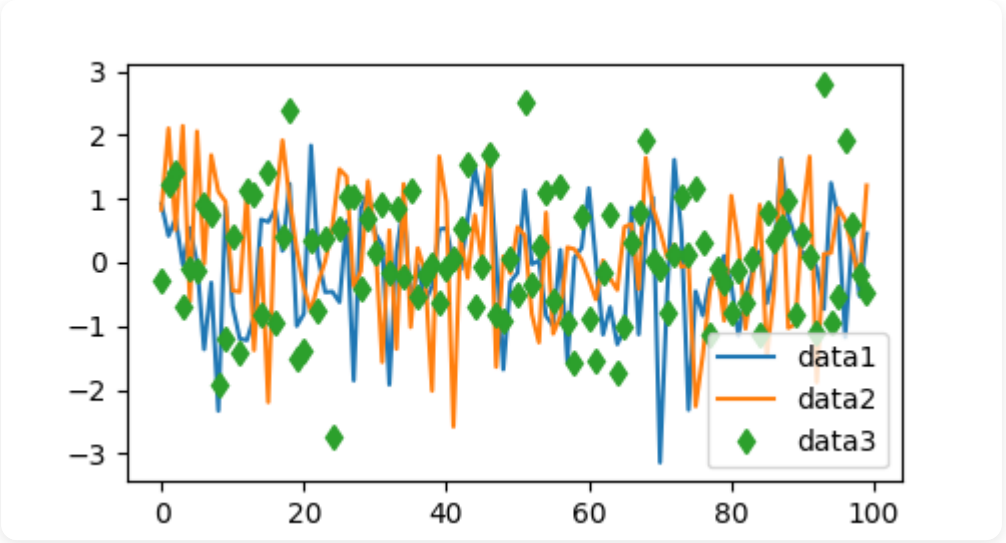
코사인 그래프의 지역 최댓값(local max) 지점을 화살표와 텍스트로 가리키는 주석.

이 기본 예제에서 `xy`와 `xytext`는 모두 데이터 좌표계에 있습니다. 이 외에도 다양한 좌표계를 선택할 수 있습니다. (자세한 내용은 **주석 가이드** 참조)

6.4. 범례 (Legends)

여러 개의 선이나 마커가 있는 그래프에서는 각각이 무엇을 나타내는지 식별하기 위해 범례(`Axes.legend`)가 필요합니다. 플로팅할 때 `label` 인자를 지정하면, `legend()` 메서드가 자동으로 해당 레이블을 사용하여 범례를 생성합니다.

```
fig, ax = plt.subplots(figsize=(5, 2.7))
ax.plot(np.arange(len(data1)), data1, label='data1')
ax.plot(np.arange(len(data2)), data2, label='data2')
ax.plot(np.arange(len(data3)), data3, 'd', label='data3')
ax.legend()

plt.show()
```



세 개의 다른 데이터 시리즈를 식별하는 범례가 추가된 그래프.

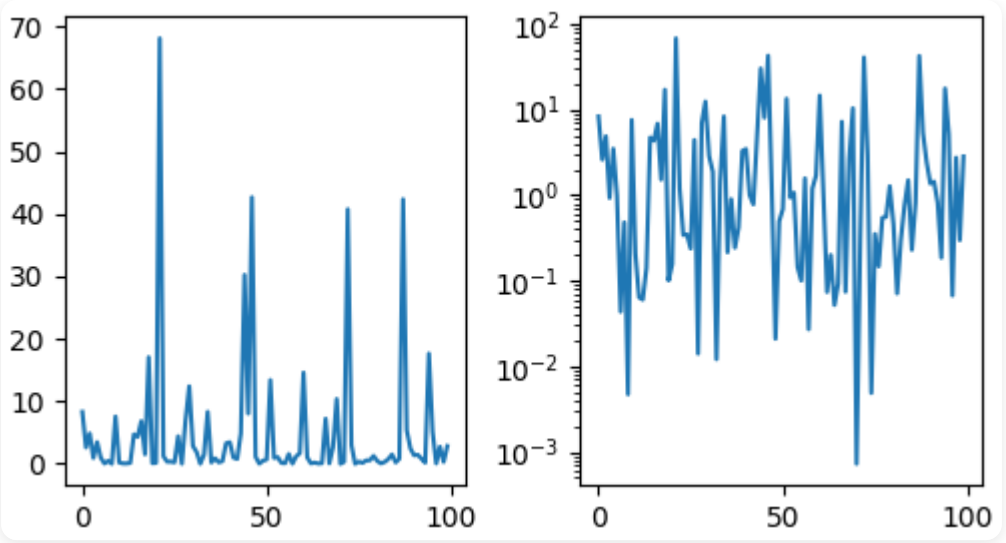
7. 축 스케일과 눈금 설정

각 Axes는 x축과 y축을 나타내는 두 개(3D의 경우 세 개)의 `Axis` 객체를 가집니다. 이 객체들은 축의 스케일, 눈금 위치(tick locators), 눈금 형식(tick formatters)을 제어합니다.

7.1. 스케일 (Scales)

선형 스케일 외에도 Matplotlib는 로그 스케일(log-scale)과 같은 비선형 스케일을 제공합니다. 로그 스케일은 매우 자주 사용되므로 `loglog`, `semilogx`, `semilogy`와 같은 직접적인 메서드도 있습니다. 아래 예제에서는 수동으로 y축 스케일을 로그로 설정합니다.

```
fig, axs = plt.subplots(1, 2, figsize=(5, 2.7), layout='constrained')
xdata = np.arange(len(data1))
data = 10**data1

# 왼쪽: 선형 스케일
axs[0].plot(xdata, data)
axs[0].set_title('Linear scale')

# 오른쪽: 로그 스케일
axs[1].plot(xdata, data)
axs[1].set_yscale('log')
axs[1].set_title('Log scale')

plt.show()
```

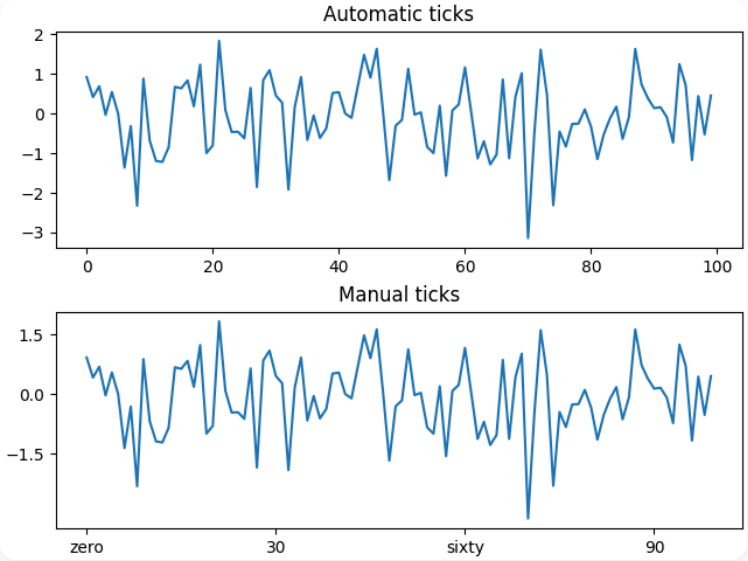


동일한 데이터를 선형 스케일(왼쪽)과 로그 스케일(오른쪽)로 표현한 그래프 비교.

7.2. 눈금 위치 지정자와 형식 지정자 (Tick Locators and Formatters)

각 축에는 눈금 표시를 어디에 둘지 결정하는 눈금 위치 지정자(locator)와 눈금 레이블 형식을 지정하는 형식 지정자(formatter)가 있습니다. 이에 대한 간단한 인터페이스는 `set_xticks`와 `set_yticks`입니다. 이 함수들을 사용하면 눈금의 위치와 레이블을 수동으로 설정할 수 있습니다.

```
fig, axs = plt.subplots(2, 1, layout='constrained')

# 위: 자동 눈금
axs[0].plot(xdata, data1)
axs[0].set_title('Automatic ticks')

# 아래: 수동 눈금
axs[1].plot(xdata, data1)
axs[1].set_xticks(np.arange(0, 100, 30), ['zero', '30', 'sixty', '90'])
axs[1].set_yticks([-1.5, 0, 1.5])  # 레이블을 지정하지 않으면 숫자 그대로 표시
axs[1].set_title('Manual ticks')

plt.show()
```



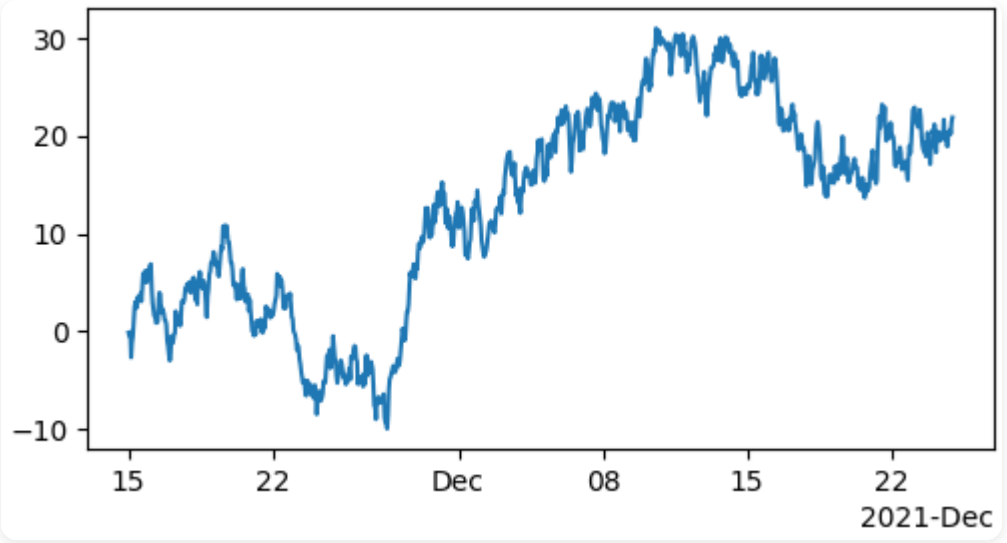
자동으로 생성된 눈금(위)과 사용자가 직접 위치와 레이블을 지정한 수동 눈금(아래) 비교.

7.3. 날짜 및 문자열 플로팅

Matplotlib는 숫자 데이터뿐만 아니라 날짜와 문자열 배열도 처리할 수 있습니다. 이러한 데이터 유형에는 적절한 위치 지정자와 형식 지정자가 자동으로 적용됩니다.

날짜 데이터

```
from matplotlib.dates import ConciseDateFormatter

fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
dates = np.arange(np.datetime64('2021-11-15'), np.datetime64('2021-12-2
                  np.timedelta64(1, 'h'))
data = np.cumsum(np.random.randn(len(dates)))

ax.plot(dates, data)
# 날짜 형식 지정자를 간결한 형태로 설정
ax.xaxis.set_major_formatter(
    ConciseDateFormatter(ax.xaxis.get_major_locator()))

plt.show()
```

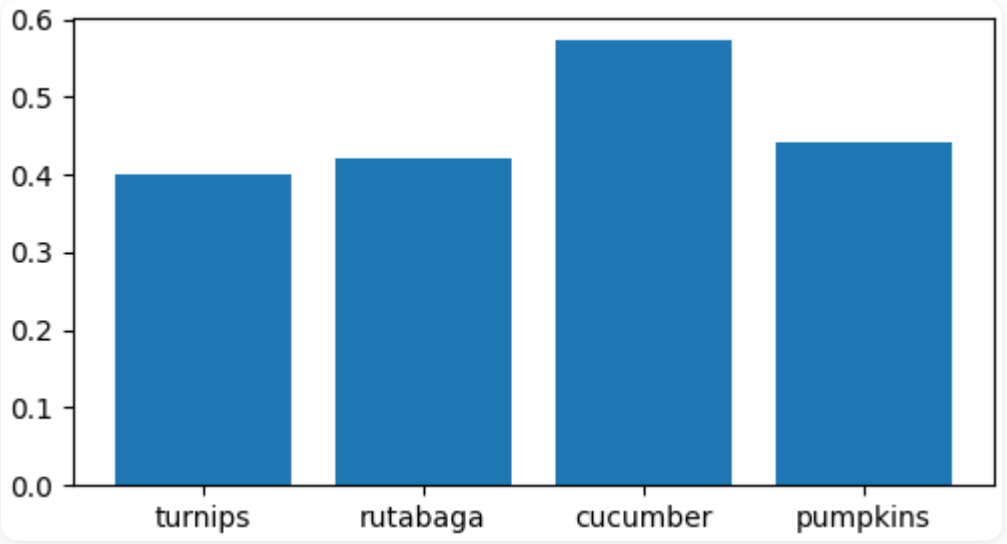


x축이 날짜 데이터로 구성된 시계열 그래프. 날짜 형식 지정자가 적용되어 있습니다.

문자열 데이터 (범주형)

x축이나 y축에 문자열 리스트를 전달하면, Matplotlib는 이를 범주형 데이터로 인식하고 각 문자열에 대해 눈금을 생성합니다.

```
fig, ax = plt.subplots(figsize=(5, 2.7), layout='constrained')
categories = ['turnips', 'rutabaga', 'cucumber', 'pumpkins']
ax.bar(categories, np.random.rand(len(categories)))

plt.show()
```



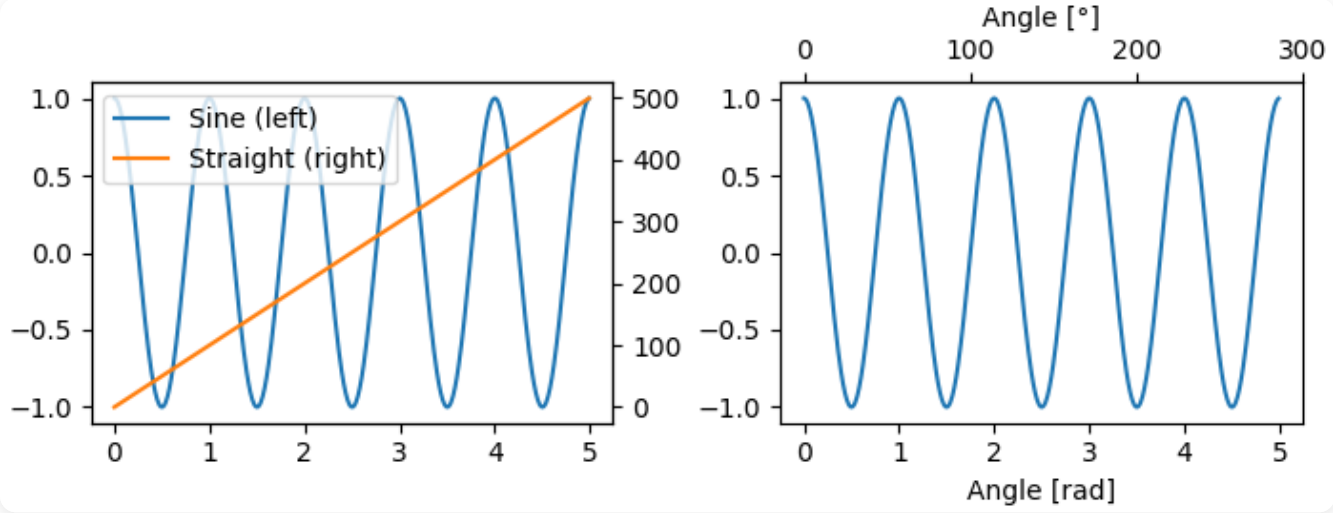
x축이 문자열 카테고리로 구성된 막대 그래프.

7.4. 추가 축 객체 (이중 축)

하나의 차트에서 크기 단위가 매우 다른 데이터를 함께 표시해야 할 경우, 추가적인 y축이 필요할 수 있습니다. `twinx()`를 사용하면 보이지 않는 x축과 오른쪽에 위치한 y축을 가진 새로운 Axes를 생성할 수 있습니다. (y축을 공유하고 x축을 분리하려면 `twiny()`를 사용합니다.)

또한, `secondary_xaxis()` 또는 `secondary_yaxis()`를 사용하여 주 축과 다른 스케일이나 단위를 가진 보조 축을 추가할 수도 있습니다. 예를 들어, 라디안(radian) 단위를 도(degree) 단위로 변환하여 함께 표시할 수 있습니다.

```
fig, (ax1, ax3) = plt.subplots(1, 2, figsize=(7, 2.7), layout='constrai

# 이중 Y축 예제
l1, = ax1.plot(t, s)
ax2 = ax1.twinx()
l2, = ax2.plot(t, range(len(t)), 'C1')
ax2.legend([l1, l2], ['Sine (left)', 'Straight (right)'])

# 보조 X축 예제
ax3.plot(t, s)
ax3.set_xlabel('Angle [rad]')
def rad2deg(x):
    return np.rad2deg(x)
def deg2rad(x):
    return np.deg2rad(x)
secax = ax3.secondary_xaxis('top', functions=(rad2deg, deg2rad))
secax.set_xlabel('Angle [°]')

plt.show()
```

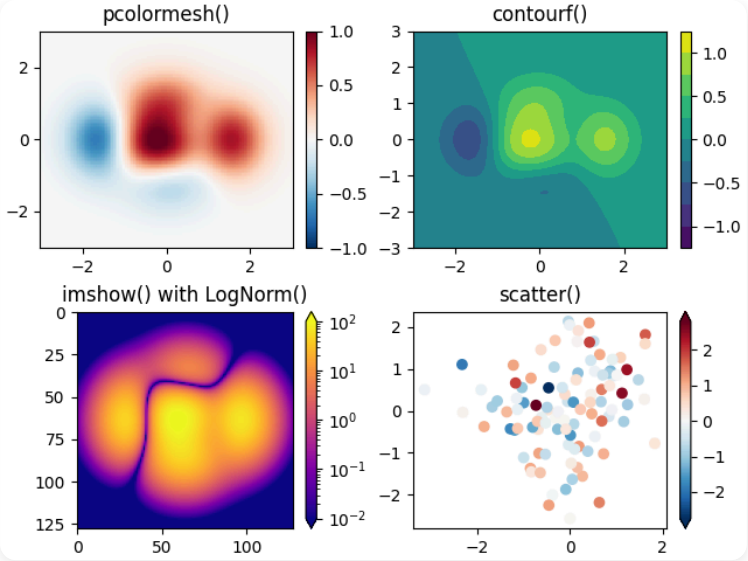


왼쪽: 서로 다른 스케일을 가진 두 개의 y축(이중 축). 오른쪽: 라디안과 각도를 함께 보여주는 보조 x축.

8. 특정 그래프 유형: 색상 매핑 데이터 (히트맵 등)

종종 2D 플롯에서 세 번째 차원을 색상으로 표현하고 싶을 때가 있습니다. Matplotlib는 이를 위한 여러 플롯 유형을 제공합니다. 대표적으로 `pcolormesh`, `contourf`, `imshow`, 그리고 색상 인자를 받는 `scatter`가 있습니다.

```
from matplotlib.colors import LogNorm

# 2D 데이터 생성
X, Y = np.meshgrid(np.linspace(-3, 3, 128), np.linspace(-3, 3, 128))
Z = (1 - X/2 + X**5 + Y**3) * np.exp(-X**2 - Y**2)

fig, axs = plt.subplots(2, 2, layout='constrained')

# pcolormesh
pc = axs[0, 0].pcolormesh(X, Y, Z, vmin=-1, vmax=1, cmap='RdBu_r')
fig.colorbar(pc, ax=axs[0, 0])
axs[0, 0].set_title('pcolormesh()')

# contourf (등고선 채우기)
co = axs[0, 1].contourf(X, Y, Z, levels=np.linspace(-1.25, 1.25, 11))
fig.colorbar(co, ax=axs[0, 1])
axs[0, 1].set_title('contourf()')

# imshow (이미지) with LogNorm
pc = axs[1, 0].imshow(Z**2 * 100, cmap='plasma',
                      norm=LogNorm(vmin=0.01, vmax=100))
fig.colorbar(pc, ax=axs[1, 0], extend='both')
axs[1, 0].set_title('imshow() with LogNorm()')

# scatter (산점도)
pc = axs[1, 1].scatter(data1, data2, c=data3, cmap='RdBu_r')
fig.colorbar(pc, ax=axs[1, 1], extend='both')
axs[1, 1].set_title('scatter()')

plt.show()
```



pcolormesh, contourf, imshow, scatter를 이용한 다양한 색상 매핑 데이터 시각화 예제.

8.1. 컬러맵 (Colormaps)

위 예제들은 모두 `ScalarMappable` 객체에서 파생된 Artist들입니다. 이들은 `vmin`과 `vmax` 사이의 데이터 값을 `cmap`으로 지정된 컬러맵에 선형적으로 매핑합니다. Matplotlib는 다양한 내장 컬러맵을 제공하며, 직접 만들거나 서드파티 패키지를 통해 다운로드할 수도 있습니다. (참조: **컬러맵 선택 가이드**)

8.2. 정규화 (Normalizations)

때로는 위 `LogNorm` 예제처럼 데이터와 컬러맵 간의 비선형 매핑이 필요할 수 있습니다. 이때는 `vmin`, `vmax` 대신 `norm` 인자를 제공하여 정규화 객체를 전달합니다. (참조: **컬러맵 정규화**)

8.3. 컬러바 (Colorbars)

`colorbar`를 추가하면 색상이 실제 데이터 값과 어떻게 관련되는지 보여주는 키(key)를 제공할 수 있습니다. 컬러바는 Figure 수준의 Artist이며, 보통 부모 Axes로부터 공간을 빌려와 배치됩니다. 컬러바의 배치는 복잡할 수 있으므로 상세한 가이드를 참고하는 것이 좋습니다. (참조: **컬러바 배치 가이드**)

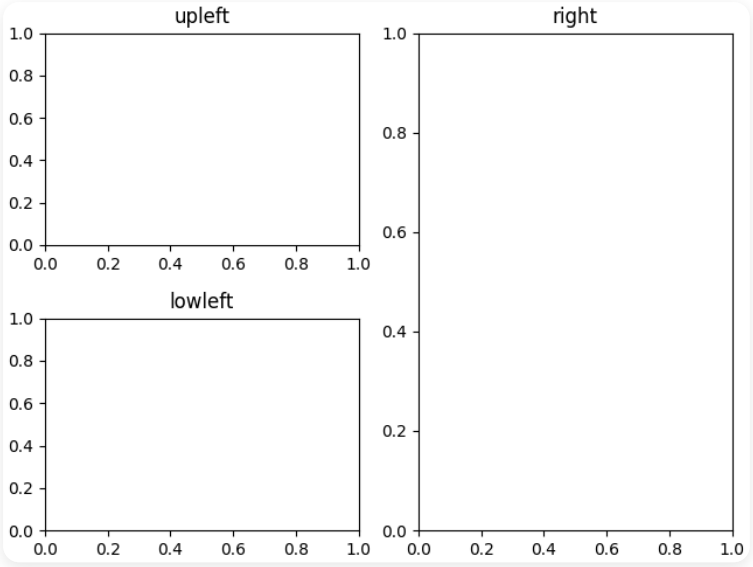
9. 여러 Figure와 Axes 다루기

`fig = plt.figure()` 또는 `fig2, ax = plt.subplots()`를 여러 번 호출하여 여러 개의 Figure를 열 수 있습니다. 각 Figure 객체 참조를 유지하면 어떤 Figure에든 Artist를 추가할 수

있습니다.

하나의 Figure에 여러 Axes를 추가하는 가장 기본적인 방법은 위에서 사용한 `plt.subplots()`입니다. 열이나 행에 걸쳐 있는 Axes 등 더 복잡한 레이아웃을 구현하려면 `subplot_mosaic`를 사용할 수 있습니다. 이 함수는 문자열로 레이아웃을 직관적으로 설계하고, 각 Axes에 이름으로 접근할 수 있게 해줍니다.

```
fig, axd = plt.subplot_mosaic([['upleft', 'right'],
                               ['lowleft', 'right']], layout='const

# 딕셔너리처럼 이름으로 각 Axes에 접근하여 제목 설정
axd['upleft'].set_title('upleft')
axd['lowleft'].set_title('lowleft')
axd['right'].set_title('right')

plt.show()
```



`subplot_mosaic`를 사용하여 생성된 비대칭적인 서브플롯 레이아웃.

10. 결론 및 추가 학습 자료

이 가이드를 통해 Matplotlib의 핵심 구성 요소, 두 가지 코딩 스타일, 그리고 그래프를 꾸미고 제어하는 다양한 방법에 대해 알아보았습니다. 이제 여러분은 Matplotlib의 기본 사용법을 숙달하고, 실제 데이터 시각화 프로젝트에 이를 적용할 준비가 되었습니다.

효율적인 데이터 시각화를 위해서는 꾸준한 연습과 다양한 예제를 접하는 것이 중요합니다. 다음 자료들을 통해 더 깊이 학습해 보시길 권장합니다.

- **다양한 플롯 유형 (Plot types)**: Matplotlib으로 그릴 수 있는 수많은 종류의 그래프 예제를 확인하세요.
- **공식 갤러리 (Gallery)**: 수백 가지의 예제 코드와 결과물을 통해 영감을 얻고 코드를 학습하세요.
- **API 참조 (API reference)**: 특히 **Axes API**는 가장 자주 사용하게 될 메서드들의 상세한 설명을 담고 있습니다.

데이터 시각화는 데이터를 이해하고 인사이트를 발견하는 강력한 도구입니다. Matplotlib와 함께 데이터 탐험의 여정을 즐기시길 바랍니다.

참조: Matplotlib 3.10.7 Documentation - Quick start guide

참고 자료

[1] https://matplotlib.org/stable/users/explain/quick_start.html
https://matplotlib.org/stable/users/explain/quick_start.html