추가

- OSS 패키지 설치 업로드
    - PyPI
        - Python 패키지를 공유하고 배포하는 공식적인 중앙 Repository
        - 전 세계 Python 개발자들이 만든 오픈 소스를 업로드하고, 다른 사람들이 설치할 수 있도록 하는 플랫폼
        - 즉, Python 생태계의 Appstore 같은 느낌

```
pip install git+https://github.com/psf/request.git # git 최신 버전 소스 직접 설치
pip install requests # PyPI에서 request 패키지를 받아 설치
# numpy, pandas, flask, sckit-learn 등이 PyPI에 있는 유명한 패키지들

# 업로드
# 패키지 설치
pip install setuptools wheel twine

# 패키지 빌드
python setup.py sdist bdist_wheel

# PyPI에 업로드
twine upload dist/*
```

PyPI(Python Package Index)

- 파이썬 프로그래밍을 위한 공식 소프트웨어 저장소
- 전 세계 개발자들이 작성한 파이썬 패키지를 등록하고 공유하는 공간
- 주요 기능
    - 패키지 검색 및 설치
    - 패키지 업로드 및 배포
    - 의존성 관리
- 라이브러리 검색, 설치 및 발행하는 곳: https://pypi.org/
- Twine을 이용해 업로드 시
    - -u __token__ -p [복사한 토큰]
- Publish를 위한 코드를 준비(이름은 Unique해야 함)

PyPI에 패키지 업로드

```
!pip install wheel
!pip install twine
!python setup.py bdist_wheel # 컴파일
!twine upload -u __token__ -p [토큰]  dist/filename # 업로드
```

- 에러
    - Pypi에 존재하는 패키지 이름과 동일한 경우 → 고유한 이름으로 변경
    - 동일한 버전을 올리는 경우 → 다른 버전으로 올리기
    - 사용자이름과 비밀번호를 사용한 경우 → Token을 사용하여야 함
- Pip을 통해 업로드된 패키지 설치

```
!pip install packagename
import mycalc # 패키지에 대한 폴더명
From mycalc import mycalc # 폴더명, 폴더 내 파일명
```

Streamlit

- 파이썬 프레임워크
- 장점
    - Scikit, Keras, Numpy, Pandas, Tensorflow와 호환
    - 개발 속도 빠름
    - 안전, 보안 지원되는 Webapp
    - No HTML, CSS, JavaScript
    - 배포가 쉬움

```
# Basic Text Elements
st.title()
st.header()
st.markdown()
st.subheader()
st.caption()
st.code()
st.latex(r''' a + a r ^1 + a r^2 + a r^3 ''')
```

Docker

- 오픈 SW 프로젝트 시, 재생산성과 접근성을 향상(Docker Image/Container, Dockerfile, DOcker-compose)
- 주요 개념
    - 프로세스 실행을 위한 환경 정의
    - 환경의 분리를 정돈
    - 정의된 리소스와 함께 샌드박스 설정
    - 응용 실행을 위한 하나의 간단한 인터페이스
- **도커가 불가능한 것: 가상화, 분리된 커널, 하이퍼바이저**
- Container는 SIngle 프로세스를 실행 가능
- Single command를 캡슐화한 Docker
- Docker container는 조합 가능(DB, Web Server, 네트워크 포트 위에 개별 서비스 제공)

- 설치: 리눅스 기반(윈도우, mac os 가능)
- Ubuntu, Debian에서 설치하는 방법

```
curl https://get.docker.com/ | sh
```

Docker-machine

- Docker 호스트를 생성하고 관리하기 위한 가상의 박스 위의 간단한 Wrapper

```
docker-machine create --driver=virtualbox dev # create new machine
docker-machine start dev
docker-machine upgrade dev # upgrade the vm image of machine 'dev'
docker-machine ls
eval $(docker-machine env dev) # activate enviroment variable to machine
docker-machine ssh dev # ssh into machine "dev"
```

커맨드 라인 인터페이스

- **중요**

```
docker run \
    --name Container's name \ # 컨테이너 이름을 설정
    --p8080:8080 -p 7080:7080 \ # 8080, 7080 포트를 컨테이너와 바인딩
    -d \ # 백그라운드 모드 실행
    -v $Home/../data:/data \ # 호스트의 디렉토리를 컨테이너 내부 /data에 마운트
    image:latest # image의 최신버전 이용
```

Images와 Containers

- **Docker Image**
  - 컨테이너를 위한 Immutable 템플릿
  - 레지스트리로부터 pull, 레지스트리에 push 할 수 있음
  - [registry/][user/]name[:tag] 형태의 이미지 이름
  - Latest는 기본 tag
- **Docker Container**
  - Image의 instance
  - Start, stop, restart 될 수 있음
  - 파일 시스템 내에서 변화를 유지함
  - 새로운 이미지는 현재 Container 상태로부터 생성될 수 있음(다만 Docker file을 사용할 것을 추천)
  - **CMD**: Container가 시작하자마자 수행되는 커맨드를 정의하는 키워드

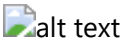
이미지 핸들링을 위한 커맨드

```
docker search term # 레지스트리 내에서 검색
docker pull 이미지명:태그 # 레지스트리로부터 이미지를 다운로드하거나 업데이트(*)
Docker push image # 레지스트리에 이미지를 업로드
docker images # 다운로드된 이미지 리스팅(*)
docker tag oldname newname # 이미지에 새로운 이름 부여
docker rmi image # 로컬에 있는 이미지 삭제
docker run imagename # 새로운 컨테이너 실행(*) -p 8080:80
docker ps # 실행중인 컨테이너 리스트(*)
docker ps -a # 모든 컨테이너 리스트
docker inspect container # 컨테이너의 메타데이터 보기
docker stop container # 컨테이너 종료(*)
docker start container # 컨테이너 시작
docker kill container # 컨테이너 강제 중단
docker remove container # 컨테이너 제거
docker exec container comman # 컨테이너 내 커맨드 실행
docker exec -it container bash # 컨테이너 내 커맨드 실행
docker logs -f container # 컨테이너 로그 보기
docker cp my_webserver:/etc/nginx/ngix.conf ~/ # 컨테이너에 파일 복사하기
docker diff container # 컨테이너 파일 시스템 내 변화 감지
docker commit container imagename # 컨테이너 변화로부터 새로운 이미지 생성
```

Docker Compose

- 컨테이너 간 응용서비스들을 설정 혹은 모든 서비스 생성(YAML 파일을 사용)
- 자동화된 멀티 컨테이너 워크플로우(개발, 테스팅, CI 워크플로우, 단계별 환경)
- 특징
    - 싱글 호스트 위에서 여러 개의 독립된 환경을 구축
    - 컨테이너가 생성되었을 때, 디스크 볼륨을 보존
    - 변화된 컨테이너만 재생성
    - 환경 간에 조합
    - 동시에 작업하도록 다양한 컨테이너를 오케스트레이션

Docker Images와 Layers



명령어

- **FROM 이미지명**: 모든 Dockerfile의 첫 명령어, Base image 기본 이미지 설정
- **RUN command**

```
RUN apt-get install -y memcached
```

- **Docker build** : Dockerfile로부터 이미지 생성

```
docker build -t memcached_d1 .
docker run -t -t memcached_d1 /bin/bash
```

예제

```
docker pull jupyter/base-notebook
docker images
docker run -dit -p 8888:8888 jupyter/base-notebook
docker ps
```

- **docker-compose.yml 예제**
  - web, redis 서비스가 정의
  - web 서비스는 현재 디렉토리의 Dockerfile 기반으로 이미지를 빌드
  - 호스트의 8080포트를 컨테이너 내부의 5000포트로 연결하여 web에서 localhost:8080으로 접근 가능
  - web 서비스는 Reids 서비스가 먼저 실행되어야함
  - redis는 Docker hub에 공식 이미지를 사용

```
version: '2' # 버전 Number
services:
    web:
        build: . # 현재 디렉토리의 Dockerfile 기반 이미지 빌드
        ports:
            - "8080:5000" # 호스트 8080 에서 컨테이너 5000 포트로 전달
        volumes:
            - .:/code
        depends_on:
            - redis # Web 서비스 실행 전 Redis가 먼저 실행되어야 함
    redis:
        image: redis # 공식 Redis 이미지 사용
```

Pytest

Testing의 주목적

- 간단, 확장 가능한 테스트 케이스
- 주로 API를 위한 테스트 코드 작성을 위해 사용되었음
- 단위 테스트 ~ 복잡한 기능 테스트까지 지원
- 부가적 목적
  - 품질 수준에 대한 자신감 획득과 정보 제공
  - 비즈니스 리스크를 감소시키는 정보에 근거한 조언
  - 개발 프로세스 점검, 이슈 제기
  - 논리적 설계의 구현을 검증

Pytest(CI/CD Tools)

- **CI/CD**: Continuous Integration/ Continuous Deployment
- 매우 간단한 Syntax
- 병렬 테스팅 가능
- 특정 테스트와 테스트의 서브셋을 실행 가능
- 자동으로 테스트 검출

어떻게 Pytest는 Test파일과 메소드를 식별하는가

- **Test에 대한 Prefix로 탐지(접두어, 접미어)**
    - test_
    - _test
- **Test 메소드는 test 키워드로 시작한 것만 인정
    - e.x. def test_file1_method1():
- 하나의 파일 혹은 여럭의 테스트 파일을 수행하는 방법
    - 여러개: py.test(하위 폴더까지 실행)
    - 하나의 파일만 실행: py.test test_sample1.py
- Marker에 의한 테스트 그룹(@pytest.mark)

Pytest Fixtures

- 매 테스트 메소드 전에 일부 코드를 반복적으로 실행하고 싶을 때,
    - **Fixtures**: 반복되는 고정된 코드
    - DB 연결 등
- @pytest.fixture

예제

```
!pip install pytest
```

- Python 파일

```
import pytest
def test_file1_method1():
    x=5
    y=6
    assert x+1 == y, "test failed"
    assert x == y, "test faile"
def test_file1_method2():
    x=5
    y=5
    assert x+1 == y, "test failed"
```