Python 표준 라이브러리

표준 라이브러리

참조: https://docs.python.org/ko/3.9/library/index.html

이전 토픽에서 공부했던 라이브러리의 의미를 다시한번 생각해본다.

라이브라리란 다양한 모듈을 포함하고있는 패키지들의 집합이다. 즉, Python 표준 라이브러리는 일상적인 프로그래밍에서 발생하는 문제에 대한 표준적인 해결책을 제공하는 다양한 모듈이 포함되어있다. 이러한 모듈은 별도의 PIP를 활용한 다운로드가 필요 없으며, Python설치시 자동으로 설치된다.

각각의 표준 모듈은 매우 다양한 기능과 함수를 내포하고 있기 때문에 하나하나 모두 짚으며 내용을 정리하기엔 무리가 있다. 따라서 각각의 모듈별로 몇가지의 기능만을 소개하며, 자세한 내용은 첨부된 공식 링크를 통하여 별도로 확인 바란다.

OS 모듈: 운영체제 인터페이스

참조: https://docs.python.org/ko/3.9/library/os.html

OS 모듈은 현재 사용하는 운영 체제에 코드로 접근하고자 할때 많이 사용된다.

예를들어 현재 운영체제의 버전 릴리스, 호스트 네임 등의 정보를 불러오거나, 파일 시스템을 탐색하거나, 환경변수에 접근하는 인터페이스를 제공해주는 모듈이다.

OS 모듈의 경우, 운영 체제 인터페이스이기 때문에 각 운영체제별로 가용성이 상이하다. 따라서 위 공식 문서에 각 함수 및 속성에 적혀있는 가용성을 참고하자.

환경 변수

`os.environ`

```
import os

# Getting all environment variables
environment_variables = os.environ  # returns a dictionary objec
print(environment_variables)
# {'HOME': '/home/user1', 'PATH': '/bin:/usr/bin'}

# Get one environment variable
my_shell = os.environ["SHELL"]  # returns a string object
print(my_shell)
# /bin/bash

# appending to environment variable
os.environ["FOO"] = "bar"  # append to environment variable
foo = os.environ["foo"]  # check if successfully appended
print(foo)
# bar
```

위와같이 환경 변수를 dictionary 형태로 접근할 수 있다.

환경 변수의 키와 값은 dictionary형태로 매핑되며, 기존에 설정되어있는 환경 변수를 확인하거나 새로운 환경 변수를 등록해줄 수 있다.

`os.getenv`

```
import os

# This will cause KeyError
print(os.environ['MY_NAME'])
```

```
# KeyError: 'MY_NAME'

# This will not cause an error
print(os.getenv('MY_NAME')
# None

# You can also set defult value, in case the environment variabl
print(os.environ('MY_NAME', 'Daniel'))
# 'Daniel'

# However, `os.getenv` does not really set the environment varia
print(os.environ['MY_NAME'])
# KeyError: 'MY_NAME'
```

가장 위에서부터 하나씩 코드를 살펴본다.

`os.environ` 의 경우 dictionary 형태로 환경 변수로 접근한다고 했다. 따라서 해당 dictionary에 없는 키 값을 찾으려고 한다면 `KeyError` 가 발생하게된다.

`os.getenv`는 dictionary의 `dictionary.get`과 동일하게 동작하며, 만약 찾고자 하는 키 값이 해당 dictionary에 없다면 `None`을 반환하고 오류를 발생시키지 않는다.

또한 `os.getenv` 함수의 두번째 인자로 기본값을 전달해줄 수 있다. 즉, 찾고자 하는 키 값이 해당 환경 변수에 등록되어있지 않다면 `None` 대신 해당 기본값을 반환해주도록 하는 것이다.

하지만 해당 기본값은 일회성으로 반환되며, 환경 변수에 설정되진 않는다. 환경 변수에 등록하여 사용하고 싶다면 `os.environ['NEW_KEY'] = 'NEW_VALUE'`와 같은 형태로 직접 등록해주어야 한다.

현재 접속중인 사용자 및 세션

`os.getlogin`

```
import os

user_name = os.getlogin()
```

```
print(user_name)
# user1
```

현재 접속중인(로그인된) 사용자의 계정명을 반환 한다.

`os.getuid` | `os.getgid`

```
import os

user_uid = os.getuid()
print(user_uid)
# 1000

user_gid = os.getgid()
print(user_gid)
# 1000
```

현재 접속중인(로그인된) 사용자의 UID를 반환 한다.

`os.getcwd`

```
import os

cwd = os.getcwd()
print(cwd)
# /home/user1
```

현재 위치한 디렉토리 (Current Working Directory)를 반환한다.

프로세스 및 시스템

`os.getpid`

```
import os
```

```
pid = os.getpid()
print(pid)
# 54321
```

현재 프로그램의 프로세스 아이디(PID)를 반환한다.

`os.getppid`

```
import os

ppid = os.getppid()
print(ppid)
# 12345
```

현재 프로그램의 부모 프로세스 아이디(PPID)를 반환한다.

`os.uname`

```
import os

uname = os.uname()
print(uname)
"""
posix.uname_result(
     sysname='Linux',
     nodename='host1',
     release='3.0.0-1111.11.1.el7.x86_64',
     version='#X XXX Wed May XX XX:XX:XX UTC 2000',
     machine='x86_64'
)
"""

# accessing to each element using index
print(uname[0])
# Linux
```

```
print(uname[1])
# host1

# accessing to each element using property
print(uname.sysname)
# Linux
print(uname.nodename)
# host1
```

현재 시스템의 정보를 반환한다.

`os.uname` 를 실행하면 namespace 또는 tuple형태로 접근할 수 있는 posix.uname_result 객체가 반환된다. 해당 객체의 원소들은 튜플과 같이 인덱스를 통하여 접근할 수 있으며, 각 속성을 통하여 접근할 수도 있다.

파일 시스템

```
tree /home/user1
> /home/user1
  | -- test.csh
```

아래 예시는 위와같은 파일 시스템을 예시로 한다.

`os.path.isdir`

```
from os import path

for _path in ['/home/user1', '/home/user2', '/home/user1/testFi
    if path.isdir(_path):
        print(f"EXIST: {_path}")
    else:
        print(f"NOT EXIST: {_path}")

# EXIST: /home/user1
```

```
# NOT EXIST: /home/user2
# NOT EXIST: /home/user1/testFile.txt
```

특정 디렉토리가 파일 시스템 내 존재하는지 확인한다. 만약 존재한다면 `True`를 그렇지 않다면 `False`를 반환한다.

찾고자 하는 경로가 디렉토리가 아니라 파일이라면 `False`를 반환한다.

`os.path.isfile`

```
from os import path

for _path in ['/home/user1', '/home/user2', '/home/user1/testFi
    if path.isfile(_path):
        print(f"EXIST: {_path}")
    else:
        print(f"NOT EXIST: {_path}")

# NOT EXIST: /home/user1
# NOT EXIST: /home/user2
# EXIST: /home/user1/testFile.txt
```

특정 파일이 파일 시스템 내 존재하는지 확인한다. 만약 존재한다면 `True`를 그렇지 않다면 `False`를 반환한다.

찾고자 하는 경로가 파일이 아닌 디렉토리라면 `False`를 반환한다.

`os.path.exists`

```
from os import path

for _path in ['/home/user1', '/home/user2', '/home/user1/testFi
    if path.exists(_path):
        print(f"EXIST: {_path}")
    else:
        print(f"NOT EXIST: {_path}")
```

```
# EXIST: /home/user1
# NOT EXIST: /home/user2
# EXIST: /home/user1/testFile.txt
```

파일 또는 디렉토리가 파일 시스템 내 존재하는지 확인한다. 파일, 디렉토리 관계없이 존재한다면 `True`를 그렇지 않다면 `False`를 반환한다.

만약 파일 시스템 내 존재하지만 해당 폴더, 디렉토리에 접근 권한이 없다면 `False`가 반환된다.

`os.stat`

```
import os

stat = os.stat('/home/user1/testFile.txt')
print(stat)
"""
os.stat_result(
    st_mode=33188,
    st_ino=7876932,
    st_dev=234881026,
    st_nlink=1,
    st_uid=501,
    st_gid=501,
    st_size=264,
    st_atime=1297230295,
    st_mtime=1297230027,
    st_ctime=1297230027
)
"""

# accessing to each element using index
print(stat[0])
# 33188
print(stat[1])
# 7876932
```

```
# accessing to each element using property
print(stat.st_mode)
# 33188
print(stat.st_ino)
# 7876932
```

특정 파일 또는 디렉토리의 상태를 stat_result 객체 형태로 반환한다. 해당 객체는 튜플과 같이 인덱스로 접근하거나 네임스페이스와 같이 속성명을 통하여 접근할 수 있다.

---

sys모듈: System 특정 파라미터와 함수

참조: https://docs.python.org/ko/3.9/library/sys.html

sys모듈은 의도하여 프로그램을 종료하거나, 프로그램 실행 시 특정 옵션 (파라미터)를 받아오는 기능을 구현할 때 많이 사용된다.

프로그램 종료

`sys.exit`

```
import sys

required_number = 5

if required == required_number:
    print("Program is running fine")
else:
    sys.exit(1)

# Process finished with exit code 1
```

위와같은 코드를 자주 보았을 것이다.

프로그램 실행 도중 원치 않은 방향으로 프로그램이 흘러간다면 해당 프로그램을 즉시 종료할 수 있는 아주 간단한 방법이다.

`sys.exit` 함수는 하나의 인자를 받을 수 있는데, 만약 전달되는 인자가 없다면 기본적으로 `exit code 0` 을 반환하게된다.

- `exit code 0` : 성공적인 종료
- `exit code >= 1` : 오류로 인한 종료

```
import sys

if required == required_number:
    print("Program is running fine")
elif required >= required_number:
    print("Requiring more than what we have")
else:
    sys.exit("This is too much than required")

# This is too much than required
# Process finished with exit code 1
```

위 예제와 같이 인자값으로 string 값을 전달해주면 프로그램을 종료함과 동시에 메시지를 출력할 수 있기 때문에 간단한 디버깅 프로세스에서 유용하게 활용해볼 수 있다.

인자값 받아오기

`sys.argv`

```
# main.py

import sys

args = sys.argv
print(args)
```

```
python3 ./main.py -foo bar -name daniel
> ['./main.py', '-foo', 'bar', '-name', 'daniel']
```

Python 프로그램을 실행할 때 해당 프로그램에 전달된 인자값을 받아온다.

받아온 인자값의 가장 첫번째 인자, 0, 은 항상 해당 Python 프로그램의 파일 명이다.

```
# main.py

import sys

args = sys.argv

name = ""
age = 0
for i in range(len(args[1:])):
    if args[i] == "-name":
        name = args[i+1]
    elif args[i] == "-age":
        age = int(args[i+1])
    else:
        continue

print(f"my name is {name}, and I am {age} years old"
```

```
python3 ./main.py -name daniel -age 20 -foo bar
> my name is daniel, and I am 20 years old
```

위와 같이 간단하게 사용자가 입력한 인자값을 파싱(parsing) 하여 처리할 수 있다.

다만, 이는 매우 간단하고, 기본적인 프로그램을 만들때 유용하며 많은 옵션 및 프로그램의 규모가 커질수록 코드 유지보수 및 관리가 힘들어질 수 있다.

만약 다양한 옵션으로 사용자의 인자값을 받아오고 싶다면 `argparse` 모듈을 활용해보도록 하자.

---

datatime 모듈: 날짜와 시간

참조: https://docs.python.org/ko/3.9/library/datetime.html

Python 프로그램을 작성하다보면 날짜와 시간을 활용해야 하는 경우가 생각보다 많다.

현재 날짜와 시간 뿐만 아니라 특정 데이터에서 해당 데이터가 생성된 시간, 수정된 시간의 데이터를 처리해야할 수도 있으며, `string` 형태로 구성된 날짜와 시간을 대상으로 년, 월, 일, 시, 분 별로 데이터를 나누고자 한다면 split 함수로 리스트를 만들고, 다시 인덱스를 참조하여 데이터를 빼내는 등 나름 복잡한 코드가 작성될 수 있다.

이러한 과정을 datetime 모듈을 활용하면 아주 간단히 처리할 수 있게 된다.

현재 시간 갖고오기

`datetime.now` | `datetime.today`

```
from datetime import datetime

today_1 = datetime.now()
today_2 = datetime.today()

print(today_1)
print(today_2)
# datetime.datetime(2024, 6, 1, 13, 43, 27, 695543)
# datetime.datetime(2024, 6, 1, 13, 43, 28, 88011)
```

위 `datetime.now`와 `datetime.today` 함수 모두 현재 시간을 반환한다. 해당 시간은 시스템의 로컬 시간을 기반으로 현재시간을 출력하며, 반환되는 객체는 (년, 월, 일, 시, 분, 초, 마이크로초) 와 같은 순서로 출력된다.

메소드가 두개로 정의되어 있다는 의미는 두 메소드가 같은 출력값을 반환하지만 차이점 또한 존재한다는 의미이다. 그 차이점을 알아보자.

```
from datetime import datetime
import pytz

timezones = [
    'America/New_York',
    'Europe/Rome',
    'Asia/Seoul',
]
```

```
times_of_globe = []

for tz in timezones:
    now = datetime.now(tz=pytz.timezone(tz))
    times_of_globe.append(now)

print(times_of_globe)
"""
[datetime.datetime(2024, 6, 1, 0, 56, 28, 988828, tzinfo=<DstTz
 datetime.datetime(2024, 6, 1, 6, 56, 28, 988848, tzinfo=<DstTz
 datetime.datetime(2024, 6, 1, 13, 56, 28, 988863, tzinfo=<DstT
"""
```

`datetime.now` 의 경우 인자값으로 timezone을 전달해줄 수 있다. timezone 인자의 자료형은 일반 문자형이 아닌 `tzinfo subclass` 객체를 전달해주어야 하므로 pytz 모듈을 별도로 import 하여 사용하였다.

위와같이 timezone을 `datetime.now`에 전달해주면 헌재 로컬에 설정된 시간 뿐만 아니라 세계 다른 시간대의 현재 시간을 갖고올 수 있다.

> 💡 pytz 모듈에서 사용할 수 있는 timezone 리스트는 `pytz.all_timezones` 속성으로 확인해볼 수 있다.

시간 데이터 파싱 1: 클래스 객체를 문자열로.

`datetime.strftime`

```
from datetime import datetime

current_time = datetime.now()

print(type(current_time))
# <class 'datetime.datetime'>

print(current_time)
```

```
# 2024-06-01 14:10:00.802435

print(str(current_time))
# 2024-06-01 14:10:00.802435
```

`datetime.now`와 `datetime.today` 모두 기본적으로 `datetime.datetime` 클래스 객체를 반환한다.

해당 객체를 문자열 형태로 얻고싶다면 `print` 함수 또는 `str` 함수를 사용하여 문자열로 변환할 수도 있다.

```
from datetime import datetime

current_time = datetime.now()

year = current_time.year
month = current_time.month
day = current_time.day
hour = current_time.hour
minute = current_time.minute
second = current_time.second
microsecond = current_time.microsecond

print(
            f"YEAR: {year}\n"
            f"MONTH: {month}\n"
            f"DAY: {day}\n"
            f"HOUR: {hour}\n"
            f"MINUTE: {minute}\n"
            f"SECOND: {second}\n"
            f"MICROSECOND: {microsecond}"
)
"""
YEAR: 2024
MONTH: 6
DAY: 1
HOUR: 14
```

```
MINUTE: 15
SECOND: 33
MICROSECOND: 935614
"""
```

위와같이 `datetime.datetime` 객체의 속성으로부터 년도, 월, 일 등의 각각의 날짜 및 시간 데이터를 개별로 갖고올 수도 있다. 하지만 위 결과에서 볼 수 있듯이 1월 ~ 9월과 같은 한개의 숫자로 이루어진 월은 하나의 숫자로만 표기된다. 만약 6월을 "6" 이 아닌 "06" 과 같이 추출하고 싶거나, 또는 2024년을 "2024" 가 아닌 "24" 로 추출하고 싶다면 어떻게 해야할까? 특정 날짜 및 시간 데이터를 추출함과 동시에 원하는 문자열을 추가하여 "06월" 또는 "24년" 과 같이 추출할 수 있는 방법은 없을까?

이러한 상황에서 원하는 대로 포메팅 하여 문자열로 추출할 수 있도록 도와주는 함수가 `datetime.strftime` 이다.

```
from datetime import datetime

now = datetime.now()

year = now.strftime("%y")
month = now.strftime("%m")
day = now.strftime("%d 일")
current_time = now.strftime("%H:%M:%S, %A")

print(year)
# 24
print(month)
# 06
print(day)
# 01 일
print(current_time)
# 14:15:33, Satueday
```

`datetime.strftime` 함수를 사용하면 손쉽게 `datetime.datetime` 객체를 원하는 형태로 포메팅하여 문자열 형태로 출력할 수 있다.

위에서 사용된 `%y`, `%m`, `%d`, `%H` 등 앞에 `%` 가 붙은 문자들은 `format codes` 라고 하여 datetime 모듈에서 특정 데이터를 지칭하는 `지시자` 라고 한다.

더 많은 `지시자`와 의미에 관련된 내용은 아래 링크를 참조하자.

참조: https://docs.python.org/ko/3.9/library/datetime.html#strftime-and-strptime-format-codes

---

시간 데이터 파싱 2: 문자열을 클래스 객체로.

`datetime.strptime`

그럼 이번엔 그 반대 경우를 생각해보자.

```
from datetime import datetime

current_time = "2024.01.01_10:05:34"
```

위와 같이 문자열 형태의 시간이 주어졌고, 이 문자열을 `datetime.datetime` 객체로 바꾸고 싶다면 어떻게 해야할까? (이러한 상황은 생각보다 더 빈번하게 일어난다. 예를 들어 위와 같은 문자열의 시간 데이터가 주어졌을 때 해당 일에 대한 요일을 찾고자 할 수도 있고, 해당 형태를 다른 형식으로 변환 하고자 할 수도 있으며, 해당 시간을 기준으로 과거 또는 미래의 시간 혹은 날짜를 계산하고자 할 수도 있다.)

```
from datetime import datetime

current_time = "2024.01.01_10:05:34"

dt_current_time = datetime.strptime(current_time, '%Y.%m.%d_%H:%
print(dt_current_time)
# datetime.datetime(2024, 1, 1, 10, 5, 34)
```

문자열로 작성된 날짜와 시간 데이터를 `datetime.strptime`함수로 전달한 후 포멧에 맞추어 지시자를 작성해주면 해당 각각의 데이터가 파싱되어 `datetime.datetime` 객체를 생성한다.

```
from datetime import datetime

current_time = "2024.01.01_10:05:34"
```

```
dt_current_time = datetime.strptime(current_time, '%Y.%m.%d_%H:%

day_of_week = dt_current_time.strftime("%A")
print(day_of_week)
# Monday

new_date = dt_curret_time.strftime('%y-%m-%d %H')
print(new_date)
# 24-01-01 10
```

생성된 `datetime.datetime` 객체를 활용하여 기존에 모르던 요일 등의 데이터를 추출하거나 출력 형태를 바꾸는 등 다양한 작업을 할 수 있다.

시간 데이터 파싱 3: 미래 또는 과거의 시간 구하기

`timedelta`

```
from datetime import datetime, timedelta

now = datetime(year=2024, month=6, day=1, hour=10, minute=30)
print(now)
# datetime.datetime(2024, 6, 1, 10, 30)

delta_time = timedelta(days=20, hours=5)
print(delta_time)
# datetime.timedelta(days=20, hours=5)

future_time = now + delta_time
print(futue_time)
# datetime.datetime(2024, 6, 21, 15, 30)

past_time = now - delta_time
print(past_time)
# datetime.datetime(2024, 5, 12, 5, 30)
```

`datetime.datetime` 객체를 생성하는건 `datetime.now` 또는 `datetime.today` 메소드를 사용하는 방법 뿐만 아니라 위 예제처럼 직접 날짜를 전달하여 생성해줄 수 있다.

이번 예제에서는 추가적으로 `datetime.timedelta` 모듈을 import 하였다. 해당 모듈은 시간의 흐름(duration)을 표현하기 위한 객체이다.

`datetime.datetime` 객체에 `datetime.timedelta` 객체를 더해주거나 빼주게 되면, 기준이 되는 시간으로 부터 미래 혹은 과거의 시간을 간편하게 구할 수 있다.