Python을 시작해 보자

직접 겪어 본 Python 장점

- 간결하고 가독성이 높은 문법 (Pseudo code 수준)
- 쉬운 디버깅 (친절한 에러 메시지, segmentation fault가 없다) -> 정신 건강에 이로움
- 활용도가 높은 표준 모듈들 (battery included)
- 과학/공학 계산에 유용한 확장 모듈들 (Numpy, Scipy, Matplotlib, Pandas, H5Py, 등)
- 컴파일 언어(C, C++, FORTRAN)와의 쉬운 결합
- 간편한 GPU 가속기 활용 (PyCUDA, PyOpenCL)
- 풍부한 기술문서, 도움말, 강의
- ***철학: 쉬운 일은 쉽게, 어려운 일도 가능하게***

단점

- Python을 먼저 배운다면 다른 컴파일 언어(C, C++)을 배우기가 훨씬 힘들 수 있겠다
- GIL(Global Interpreter Lock) 때문에 멀티코어 CPU를 활용하기가 번거롭다.
  - (multiprocessing이나 mpi4py를 사용해야 함)
- 모바일 컴퓨팅에는 적합하지 않은 듯 (kivy?)

Python은 실제로 많이 사용하는가?

TIOBE Index (http://www.tiobe.com/tiobe-index/)는 한 달에 한 번씩 프로그래밍 언어들의 순위를 집계한다. 평가 방식은 언어별로 숙달된 프로그래머 수, 서드파티 벤더 수, 인기있는 검색엔진(Google, Bing, Wikipedia, Amazon, Youtube, Baidu 등)에서의 사용빈도를 사용한다.

2016년까지의 TIOBE Index의 연도별 순위는 다음과 같다. Python은 이미 주류 프로그래밍 언어 중의 하나이다.

Programming Language	**2016**	**2011**	**2006**	**2001**	**1996**	**1991**	**1986**
Java	1	1	1	3	14	-	-
C	2	2	2	1	1	1	1
C++	3	3	3	2	2	2	5
C#	4	5	6	11	-	-	-
Python	**5**	**6**	**7**	**24**	**23**	**-**	**-**
PHP	6	4	4	8	-	-	-
JavaScript	7	9	8	7	19	-	-
Visual Basic .NET	8	30	-	-	-	-	-
Perl	9	8	5	4	3	-	-
Ruby	10	10	18	32	-	-	-
Lisp	27	12	12	15	7	5	3
Ada	28	16	15	16	6	3	2

Python 간단한 역사

- Python 창시자 - 네덜란드 출신의 컴퓨터 프로그래머 Guido van Rossum(1956~)
- 귀도는 ABC 언어를 개선하려는 생각을 가지고 있다가, 1989년 크리스마스 휴가 동안 파이썬 초기버전을 만들었다고 함
- 1991년 - 버전 0.9.0 공개
- 1994년 - 버전 1.0, comp.lang.python에 파이썬 포럼이 만들어져서 파이썬 사용자층이 확장됨
- 2000년 - 버전 2.0
- 2008년 - 버전 3.0, 근본적인 설계 결함을 수정하기 위해 하위 호환성을 포기함
- 현재 - 버전 2.7과 3.6이 공존, 2.7 버전은 2020년에 퇴역할 예정

Python 환경 구축

Python을 설치한다는 것은 단순히 Python 표준 모듈을 설치하는 것 뿐만 아니라, 다양한 분야의 방대한 확장 모듈들의 설치도 포함한다. 많은 확장 모듈들은 순수한 Python만으로 작성되어 있지 않고, C, C++ 등의 컴파일 언어로 작성된 라이브러리에 의존하는 경우가 많다. 외부 라이브러리 의존성은 Python 모듈 설치를 힘들게 하고, 버전 관리 또한 복잡하게 한다. 따라서 현재는 라이브러리 의존성을 확인하여 Python 모듈 설치를 쉽게 해주는 Python 배포판들을 사용한다. 주요 배포판들은 다음과 같다.

- **Anaconda**- Continuum 사에서 제공하는 배포판으로 현재 가장 널리 사용되고 있다. Python 코드 뿐만 아니라 의존하는 라이브러리들도 모두 포함하고 있어서 버전 관리가 깔끔한 것이 장점이다. 여러 개의 Python 버전을 함께 사용할 수 있는 가상 환경도 제공한다.
- Canopy - Enthought 사에서 제공하는 배포판. 상용으로 패키지 관리가 매우 잘되는 편이지만 Anaconda 출시 이후로 입지가줄어들었다.
- PythonXY - 과학/공학 계산에 특화된 배포판. 왠만한 확장 모듈을 모두 기본 설치해줘서 편하지만, 무겁고 버전관리가 어려운 단점이 있다.
- Sage - Maple, Mathematica, MATLAB 등의 상용 프로그램들의 대안으로 개발되었고, Jupyter notebook과 IPython 인터페이스를 이용하여 수치해석, 심볼릭연산 등을 위한 통합 환경을 제공한다.

Python 2 or 3?

Python 3 버전은 근본적인 설계 결함을 수정하기 위해 하위 호환성을 포기하고 새로 만들어진 버전이다. 초반에는 Python 3에 추가된 새로운 기능들은 대부분 Python 2.7.x 버전에서도 사용할 수 있도록 하였지만, 현재는 3 버전에서만 가능한 기능들이 점점 많아지고 있다.

예전에는 많은 확장 모듈들이 3 버전을 지원하지 않아서 호환성을 위해 2 버전을 계속 사용해야 했지만, 현재는 거의 대부분의 확장 모듈들이 3 버전으로 완전히 옮겨졌다. 더구나 Python 개발자 그룹에서 2.7.x 버전은 2020년까지만 지원하고 그 이후에는 개발을 중단한다고 공지하였다.

어떤 버전으로 시작해야 할지 잘 모르겠다면, 고민하지 말고 3 버전을 사용하면 된다.

대화형 인터프리터(Interactive Interpreter) 사용

IPython은 강력한 대화형 인터프리터를 제공한다. 주요 특징들은 다음과 같다.

- Python 명령들을 쉽게 확인 가능
- 외부 Python 파일을 실행하고 변수 값을 확인 가능 (디버깅 강추)
- 쉘 명령도 가능
- 병렬 환경도 지원
- notebook 형식으로 문서와 코드 통합도 가능 (이 문서와 같이)

실행 방법 - 쉘에서 ipython 명령

```
$ ipython
Python 3.6.0 |Anaconda 4.3.0 (64-bit)| (default, Dec 23 2016, 11:57:41)
Type "copyright", "credits" or "license" for more information

Ipython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of Ipython's features.
%quickref -> Quick refrence.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: print('Hello, Python!')
Hello, Python!
```

Python 코드(script) 실행

방법1) aaa.py 라는 이름의 Python 파일이 있다면 쉘에서 다음과 같이 실행하면 된다.

- $ python aaa.py

방법2) 만약 실행 파일로 만들고 싶다면,

1. Python 코드 맨 윗줄에 다음 내용을 추가하고
   - #!/usr/bin/env python
2. 쉘에서 파일 속성을 실행가능으로 변경한 후
   - $ chmod 755 aaa.py
3. 쉘에서 실행하면 된다.
   - $ ./aaa.py

방법3) IPython에서 interactive mode로 실행하고 싶다면,

- In [1]: %run aaa.py

Hello, Python

In [1]:
```
print('Hello, Python!')
```

```
Hello, Python!
```

버전 확인

In [2]:
```
import sys
print(sys.version)
```

```
3.6.0 |Anaconda 4.3.0 (64-bit)| (default, Dec 23 2016, 11:57:41) [MSC v.1900 64 bit (AMD64)]
```

들여쓰기(Indentation)

Python을 처음 접할 때 의외로 호불호가 나뉘는 부분이다.

C, C++, Perl, Java 등에서는 문단을 구분할 때 대괄호 {와 }를 사용하고, FORTRAN, Pascal에서는 문단 끝에 END 구문을 사용한다.

반면에 Python은 문단의 구문을 들여쓰기로 한다. 동일한 크기로 들여쓰기가 되어 있으면 그것은 동일 문단으로 본다. 들여쓰기는 tab이 아닌 space 4칸으로 쓰는 것을 권장한다.

In [3]:
```
sample = {'number': [1, 2, 3, 4, 5],
          'animal': ['dot', 'cat', 'pig'],
          'fruit': ['apple', 'banana', 'pear', 'coconut']}

for name, group in sample.items():
    print('[{}]'.format(name))

    for item in group:
        print(item, end=' ')

    print('₩n')
```

```
[number]
1 2 3 4 5

[animal]
dot cat pig

[fruit]
apple banana pear coconut
```

기본 자료형

- Numbers
  - Integer
  - float (double precision, 8byte, 64bit)
  - complex
- Strings

In [4]:
```
# integer
a = 5
print(type(a))

# float
b = 3.2
print(type(b))

# complex numbers
x = 3 + 4j
y = 2 - 3j
z = x + y
print(type(z), z)
```

```
<class 'int'>
<class 'float'>
<class 'complex'> (5+1j)
```

컨테이너 타입

여러 개의 변수 또는 다른 컨테이너들을 담을 수 있다.

- List []
- Tuple ()
- Dictionary {}
- set

In [5]:
```
# List
a = [1, 2, 'abc', ['d', 5]]
print(type(a))

# Tuple
b = (1, 2, 'abc', ('d', 5))
print(type(b))

# Dictionary
c = {'a':101, 'b':7, 'c':'five'}
print(type(c))

# set
d = set(['dog', 'cat', 'pig', 'cow', 'chicken'])
print(type(d))
```

```
<class 'list'>
<class 'tuple'>
<class 'dict'>
<class 'set'>
```

계산기 프로그램 실습

간단한 프로그램을 만들면서 Python 문법을 익혀보자.

두 개의 숫자와 하나의 연산자(+, -, , /, **)를 입력받아 결과를 출력하는 계산기를 만들어 보자.

버전 1

In [6]:
```
a = input('first number: ')
b = input('second number: ')
op = input('operator(+, -, *, /, **): ')
result = None

print('{} {} {} = {}'.format(a, op, b, result))
```

```
first number: 3
second number: 5
operator(+, -, *, /, **): +
3 + 5 = None
```

키보드로부터의 표준입력 함수는 input() 함수를 사용한다.

표준출력 함수는 print()이다. 문자열의 format() 내장함수를 이용하면 편리하다. 대괄호 {}는 format() 함수의 인자들을 순서대로 매칭시켜 준다.

버전 1-1

조건 분기문(if,else)을 이용하여 제대로 결과값이 나오도록 만들어보자.

In [7]:
```
a = input('first number: ')
b = input('second number: ')
op = input('operator(+, -, *, /, **): ')

if op == '+':
    result = a + b
elif op == '-':
    result = a - b
elif op == '*':
    result = a*b
elif op == '/':
    result = a/b
elif op == '**':
    result = a**b
else:
    print("'{}' operator is not supported.".format(op))
    result = None

print('{} {} {} = {}'.format(a, op, b, result))
```

```
first number: 3
second number: +
operator(+, -, *, /, **): 5
'5' operator is not supported.
3 5 + = None
```

3 + 5 = 35 와 같은 이상한 결과가 나오거나 에러가 발생할 것이다. a, b, op 변수의 타입을 출력해보라. string 타입으로 나올 것이다. input() 함수의 반환값은 문자열이므로 숫자로 사용할 때는 자료형을 변경해야 한다.

버전 1-2 (디버깅)

입력값의 자료형을 float 타입으로 변경해 보자.

In [8]:
```
a = float(input('first number: '))
b = float(input('second number: '))
op = input('operator(+, -, *, /, **): ')

if op == '+':
    result = a + b
elif op == '-':
    result = a - b
elif op == '*':
    result = a*b
elif op == '/':
    result = a/b
elif op == '**':
    print('**')
    result = a**b
else:
    print("'{}' operator is not supported.".format(op))
    result = None

print('{} {} {} = {}'.format(a, op, b, result))
```

```
first number: 3
second number: 5
operator(+, -, *, /, **): +
3.0 + 5.0 = 8.0
```

몇 가지 경우에 테스트를 해보니 이제 제대로 동작을 하는 것 같다.

앞으로 이 코드를 발전시켜 가면서 Python의 기능들을 체험해 볼 텐데, 코드 안정성과 생산성을 높이기 위해 **테스트 자동화**를 먼저 하고 가겠다. 코드를 조금씩 변경시킬 때마다 가능한 경우의 수를 일일이 확인하는 것은 너무 번거로운 일이다. 그렇다고 몇 개의 테스트만을 수행하고 지나갈 경우에는, 기존에 잘 되던 케이스가 어떤 버전부터는 동작하지 않을 수 있고, 버전을 거슬러 올라가며 디버깅을 하는 것은 정신건강에 무척 해롭다.

버전 2 (함수)

테스트 자동화를 위해 일단 함수(function)를 이용하여 코드를 정리하자.

In [9]:
```
 1 # filename: mycalc.py
 2
 3 def calc_two_terms(a, b, op):
 4     if op == '+':
 5         result = a + b
 6     elif op == '-':
 7         result = a - b
 8     elif op == '*':
 9         result = a*b
10     elif op == '/':
11         result = a/b
12     elif op == '**':
13         result = a**b
14     else:
15         print("'{}' operator is not supported.".format(op))
16         result = None
17
18     return result
19
20
21 def main():
22     a = float(input('first number: '))
23     b = float(input('second number: '))
24     op = input('operator(+, -, *, /, **): ')
25
26     result = calc_two_terms(a, b, op)
27     print('{} {} {} = {}'.format(a, op, b, result))
28
29
30 if __name__ == '__main__':
31     main()
```

```
first number: 3
second number: 5
operator(+, -, *, /, **): +
3.0 + 5.0 = 8.0
```

Python 함수 정의는 **def** 로 한다. def 라인 맨 뒤에 **:**를 적어주는 것과 **문단 들여쓰기**에 주의하라.

1번 라인에서 정의한 calc_two_terms() 함수는 두 숫자와 연산자를 받아서 계산 결과값을 반환하는 함수이고, 20번 라인에서 정의한 main() 함수는 사용자 입력을 받고 calc_two_terms() 함수를 호출하여 그 결과를 출력하는 함수이다.

29번,30번 라인은 이 Python 코드가 직접 실행될 때만 main() 함수를 실행시키라는 의미이다. 이 파일이 다른 Python 코드에서 모듈로 호출되어 사용될 때는 main() 함수가 실행되지 않는다.

버전 2-1 (테스트 자동화)

이 코드의 핵심 부분인 calc_two_terms() 함수를 테스트하는 테스트 파일을 만들어 보자. 기존의 파일은 mycalc.py 이름으로 저장하고, 테스트 파일은 test_mycalc.py 이름으로 저장한다.

In [10]:
```python
# filename: test_mycalc.py

from mycalc import calc_two_terms

def test_calc_two_terms():
    assert calc_two_terms(3, 5, '+') == 8
    assert calc_two_terms(3, 5, '-') == -2
    assert calc_two_terms(3, 5, '*') == 15
    assert calc_two_terms(3, 5, '/') == 0.6
    assert calc_two_terms(3, 5, '**') == 243
    assert calc_two_terms(3, 5, '$') == None
```

새로 만든 테스트 파일 안에 정의된 테스트 함수 test_calc_two_terms()는 calc_two_terms() 함수에 몇 가지 입력값을 주고 예상되는 결과를 확인한다. assert는 뒤의 구문이 참이면 패스하고, 거짓이면 에러를 발생시킨다. 이후에 이 함수에서 에러가 발생할 때마다 그 케이스를 추가시켜 주어서 테스트를 더 견고하게 만들어 주어야 한다.

테스트 자동화를 위해 널리 사용되는 도구는 pytest이다. 쉘에서 다음과 같은 명령으로 실행시켜 준다.

$ pytest

마지막 라인에 1 passed 라는 메시지가 나오면 테스트가 에러 없이 수행된 것이다.

pytest는 지정된 디렉토리와 그 하위 디렉토리에서 **'test'** 문자열이 포함된 파일을 탐색하고, 그 안에 정의된 **'test'** 문자열이 포함된 함수를 모두 탐색하여 실행한 후 그 결과를 한 번에 출력해 준다. 지금 예제는 파일이 하나 뿐이지만 여러 개의 파일 혹은 디렉토리 구조에서는 더욱 강력한 테스트 도구이므로 적극적으로 활용하기를 추천한다.

모듈 임포트(import)

테스트 파일 위쪽에 **from mycalc import calc_two_terms** 구문은 mycalc.py 파일 안에 정의된 calc_two_terms() 함수를 임포트(import)한 것이다. Python 코드들은 그 자체가 실행 파일이 될 수도 있고, 동시에 다른 코드에서 임포트하는 모듈이 될 수도 있다. 다른 언어에서의 라이브러리와 유사한데 훨씬 간결하고 편리하다. 만약 임포트하고 싶은 모듈 파일이 현재 디렉토리에 있다면 예제처럼

- from **(파일이름)** import **(객체이름)**

형식으로 적어주면 된다.

만약 모듈 파일이 하위 디렉토리에 있다면 **__init__.py** 파일을 추가해주어야 한다. 예를 들어, mycalc.py 파일이 아래와 같이 mod 디렉토리 밑에 위치해 있다고 하면,

- from **mod.mycalc** import **calc_two_terms**

와 같이 mod 디렉토리를 모듈 경로로 추가해주면 된다.

```
./mod/mycalc.py
./test_mycalc.py
./__init__.py (추가)
```

(참고) 만약 모듈 파일이 아예 다른 경로에 위치해 있다고 하면, PYTHONPATH 환경변수에 그 경로를 추가해 주거나, sys.path 리스트에 그 경로를 추가해 주면 된다.

버전 3 (Dictionary 사용)

이제 테스트 자동화가 구비됐으니 본격적으로 코드를 변경시켜 보자. Dictionary는 Python 기본 컨테이너 중에서도 그 쓰임새가 광활하다. key값을 포함하고 있어서 작은 데이터베이스처럼 사용할 수 있는데, 내부적으로 해쉬 테이블 기반이어서 탐색속도도 빠르다. 현재 if-else로 구분되어 있는 연산자 구분을 Dictionary로 바꾸어 보자.

In [11]:

```python
# filename: mycalc.py

def calc_two_terms(a, b, op):
    ops = {'+':a+b, '-':a-b, '*':a*b, '/':a/b, '**':a**b}

    if op in ops:
        result = ops[op]
    else:
        print("'{}' operator is not supported.".format(op))
        result = None

    return result


def main():
    a = float(input('first number: '))
    b = float(input('second number: '))
    op = input('operator(+, -, *, /, **): ')

    result = calc_two_terms(a, b, op)
    print('{} {} {} = {}'.format(a, op, b, result))


if __name__ == '__main__':
    main()
```

```
first number: 3
second number: 5
operator(+, -, *, /, **): =
'=' operator is not supported.
3.0 = 5.0 = None
```

주의) 변경된 calc_two_terms() 함수가 여전히 모든 테스트를 통과하는지 수시로 pytest 확인을 해야 한다.

버전 3-1 (더 간결하게)

calc_two_terms() 함수 있는 print() 구문을 main() 함수로 옮기고, Dictionary의 get() 내장함수를 이용해서 더 간결하게 바꿔보자. get() 내장함수는 만약 첫 번째 인자로 주어진 key 값이 없으면, 두 번째 인자를 반환한다.

- 문법: dict.get(key, alternative_value)

In [12]:
```python
# filename: mycalc.py

def calc_two_terms(a, b, op):
    ops = {'+':a+b, '-':a-b, '*':a*b, '/':a/b, '**':a**b}
    return ops.get(op, None)


def main():
    a = float(input('first number: '))
    b = float(input('second number: '))
    op = input('operator(+, -, *, /, **): ')

    result = calc_two_terms(a, b, op)
    if result == None:
        print("'{}' operator is not supported.".format(op))
    else:
        print('{} {} {} = {}'.format(a, op, b, result))


if __name__ == '__main__':
    main()
```

```
first number: 3
second number: 5
operator(+, -, *, /, **): +
3.0 + 5.0 = 8.0
```

버전 3-2 (3+0 에러)

calc_two_terms()가 간결해져서 좋지만 여기에는 함정이 있다. ops 딕셔너리가 모든 연산을 미리 해 놓는 것이 문제가 될 수 있다. 만약 **3 + 0** 입력이 들어오면 결과는 3 이어야 하지만, ops 딕셔너리에서 3/0 연산도 미리 하는 하는 바람에 에러가 발생한다.

에러 케이스를 새로 발견했기 때문에 test_mycalc.py에 새로 추가해 놓는다.

In [13]:
```python
# filename: test_mycalc.py

from mycalc import calc_two_terms

def test_calc_two_terms():
    assert calc_two_terms(3, 5, '+') == 8
    assert calc_two_terms(3, 5, '-') == -2
    assert calc_two_terms(3, 5, '*') == 15
    assert calc_two_terms(3, 5, '/') == 0.6
    assert calc_two_terms(3, 5, '**') == 243
    assert calc_two_terms(3, 5, '$') == None
    assert calc_two_terms(3, 0, '+') == 3
```

12번 라인에 새로 추가하였다. pytest 확인을 하면 에러가 발생한다.

이 문제를 해결하기 위해 operater 모듈을 사용해 보자. operator 모듈을 사용하면 문자열로 표시된 연산자를 함수로 맵핑할 수 있다.

In [15]:
```python
# filename: mycalc.py

import operator


def calc_two_terms(a, b, op):
    ops = {'+':operator.add,
           '-':operator.sub,
           '*':operator.mul,
           '/':operator.truediv,
           '**':operator.pow}
    return ops[op](a,b) if op in ops else None


def main():
    a = float(input('first number: '))
    b = float(input('second number: '))
    op = input('operator(+, -, *, /, **): ')

    result = calc_two_terms(a, b, op)
    if result == None:
        print("'{}' operator is not supported.".format(op))
    else:
        print('{} {} {} = {}'.format(a, op, b, result))


if __name__ == '__main__':
    main()
```

```
first number: 3
second number: 5
operator(+, -, *, /, **): +
3.0 + 5.0 = 8.0
```

버전 4 (간단한 입력)

이제 main() 함수에서 숫자 두 개와 연산자를 입력받는 부분을 좀 더 간단하게 바꿔보자. Python의 string 타입은 split() 내장함수를 가지고 있어서 띄어쓰기로 숫자와 연산자를 다음과 같이 구분할 수 있다.

In [16]:
```python
s = '3 + 5'
print(s.split())
```

```
['3', '+', '5']
```

In [19]:

```
##### filename: mycalc.py

import operator


def calc_two_terms(a, b, op):
    ops = {'+':operator.add,
           '-':operator.sub,
           '*':operator.mul,
           '/':operator.truediv,
           '**':operator.pow}
    return ops[op](a,b) if op in ops else None


def get_two_terms():
    expr = input('num1 op num2: ')
    sa, op, sb = expr.split()
    a = float(sa)
    b = float(sb)

    return a, b, op


def main():
    a, b, op = get_two_terms()
    result = calc_two_terms(a, b, op)

    if result == None:
        print("'{}' operator is not supported.".format(op))
    else:
        print('{} {} {} = {}'.format(a, op, b, result))


if __name__ == '__main__':
    main()
```

```
num1 op num2: 3 + 5
3.0 + 5.0 = 8.0
```

이전 버전에서 3 개의 입력을 받은 부분을 1번의 입력으로 바꾸고, 15번 라인에 get_two_terms() 함수로 바꾸었다.

17번 라인에서 split() 내장함수를 이용하여 입력 문자열을 스페이스를 기준으로 세 개의 문자열로 나누었다.

18번, 19번 라인에서 sa, sb 문자열을 float 타입으로 변경해야 하는 것에 주의하라.

버전 4-1 (예외 처리)

입력은 더 간단해졌지만 입력이 잘못된 경우 에러가 날 확률이 높아졌다. 실수로 숫자와 연산자 사이에 스페이스를 빼 먹었다면 에러가 나고 프로그램이 종료된다. Python의 예외처리 기능을 사용하여 입력에 에러가 발생했을 경우 정상적인 입력값이 들어올 때까지 반복하라고 변경해 보자.

In [20]:

```python
# filename: mycalc.py

import operator


def calc_two_terms(a, b, op):
    ops = {'+':operator.add,
           '-':operator.sub,
           '*':operator.mul,
           '/':operator.truediv,
           '**':operator.pow}
    return ops[op](a,b) if op in ops else None


def get_two_terms():
    while (True):
        try:
            expr = input('num1 op num2: ')
            sa, op, sb = expr.split()
            a = float(sa)
            b = float(sb)
        except:
            print('Wrong input, try agrain!')
        else:
            break

    return a, b, op


def main():
    a, b, op = get_two_terms()
    result = calc_two_terms(a, b, op)

    if result == None:
        print("'{}' operator is not supported.".format(op))
    else:
        print('{} {} {} = {}'.format(a, op, b, result))


if __name__ == '__main__':
    main()
```

```
num1 op num2: 3+5
Wrong input, try agrain!
num1 op num2: 3 +5
Wrong input, try agrain!
num1 op num2: 3 + 5
3.0 + 5.0 = 8.0
```

Python의 예외처리는 **try-except-(else)-(finally)** 구문으로 이루어져 있다. try 코드에서 에러가 발생하면 except 코드가 실행되고, 에러가 없으면 else 코드가 실행된다. finally 코드는 마지막에 항상 실행된다. else와 finally는 생략 가능하다. 예제에서는 except에 에러를 명시하지 않았지만, 여러 개의 except 구문에 에러를 명시하여 각각 다르게 처리할 수도 있다.

16번 라인의 while 문은 조건이 참이므로 계속 루프를 실행한다. 24번,25번 라인에서 에러가 발생하지 않으면 break 명령으로 while 루프가 중단된다.

버전 4-2 (0 나누기 예외 처리)

내친 김에 0 으로 나눌 때 발생하는 에러도 예외처리를 해보자. 이번엔 정확한 에러 이름(ZeroDivisionError)으로 에러 처리를 할 수 있다.

In [22]:

```python
# filename: mycalc.py

import operator


def calc_two_terms(a, b, op):
    ops = {'+':operator.add,
           '-':operator.sub,
           '*':operator.mul,
           '/':operator.truediv,
           '**':operator.pow}
    return ops[op](a,b) if op in ops else None


def get_two_terms():
    while (True):
        try:
            expr = input('num1 op num2: ')
            sa, op, sb = expr.split()
            a = float(sa)
            b = float(sb)
        except:
            print('Wrong input, try agrain!')
        else:
            break

    return a, b, op


def main():
    while (True):
        try:
            a, b, op = get_two_terms()
            result = calc_two_terms(a, b, op)
        except ZeroDivisionError:
            print('Zero division error, try again!')
        else:
            break

    if result == None:
        print("'{}' operator is not supported.".format(op))
    else:
        print('{} {} {} = {}'.format(a, op, b, result))


if __name__ == '__main__':
    main()
```

```
num1 op num2: 3/0
Wrong input, try agrain!
num1 op num2: 3 / 0
Zero division error, try again!
num1 op num2: 3 / 5
3.0 / 5.0 = 0.6
```

버전 5 (Tkinter GUI)

Tk는 Tcl 스크립트 언어의 GUI 확장을 위해 1991년에 처음 개발되었다. 통상 둘을 묶어서 Tk/Tcl 이라고 부르기도 한다. Tk는 당시 다른 GUI 도구들에 비해 배우기가 쉬워서 널리 사용되게 된다. 그래서 Perl, Ada, Python, Ruby 등 많은 언어들이 Tk를 사용할 수 있는 인터페이스를 장착하였다. Python에서는 tkinter(Tk interface) 이름으로 표준 모듈에 포함되어 있다.

Python에는 Tkinter 말고도 wxPython, QT, GTK+ 등 더 기능이 많은 확장 모듈들이 있지만, 간단한 GUI는 Tkinter로도 쓸만하다. 앞에서 만든 계산기를 Tkinter로 GUI 확장을 해보자.

출처: https://github.com/justudin/simple-calculator-tkinter (https://github.com/justudin/simple-calculator-tkinter)

In [23]:

```
import operator
from tkinter import Tk, messagebox
from tkinter import Button, StringVar, BOTH, LEFT, X, N
from tkinter.ttk import Frame, Label, Entry


class App(Frame):
    def __init__(self, parent):
        Frame.__init__(self, parent)
        self.parent = parent

        self.num1 = StringVar()
        self.num2 = StringVar()
        self.res = StringVar()

        self.initUI()


    def initUI(self):
        self.parent.title("Simple Calculator")
        self.pack(fill=BOTH, expand=True)

        # 4 frames vertically
        frames = []
        for _ in range(4):
            frame = Frame(self)
            frame.pack(fill=X)
            frames.append(frame)

        # two input numbers and result
        for i in [0, 1, 3]:
            txt = {0:'Number 1 :', 1:'Number 2 :', 3:'Result'}[i]
            var = {0:self.num1, 1:self.num2, 3:self.res}[i]

            label = Label(frames[i], text=txt, width=10)
            label.pack(side=LEFT, padx=5, pady=5)

            entry = Entry(frames[i], textvariable=var)
            entry.pack(fill=X, padx=5, expand=True)

        # 5 arithmetic operators
        btn1 = Button(frames[2], text='+', width=4,
                      command=lambda:self.calc_two_terms('+'))
        btn1.pack(side=LEFT, anchor=N, padx=5, pady=5)

        btn2 = Button(frames[2], text='-', width=4,
                      command=lambda:self.calc_two_terms('-'))
        btn2.pack(side=LEFT, anchor=N, padx=5, pady=5)

        btn3 = Button(frames[2], text='*', width=4,
                      command=lambda:self.calc_two_terms('*'))
        btn3.pack(side=LEFT, anchor=N, padx=5, pady=5)

        btn4 = Button(frames[2], text='/', width=4,
                      command=lambda:self.calc_two_terms('/'))
        btn4.pack(side=LEFT, anchor=N, padx=5, pady=5)

        btn5 = Button(frames[2], text='**', width=4,
                      command=lambda:self.calc_two_terms('**'))
        btn5.pack(side=LEFT, anchor=N, padx=5, pady=5)


    def calc_two_terms(self, op):
        a = float(self.num1.get())
        b = float(self.num2.get())

        ops = {'+':operator.add,
               '-':operator.sub,
               '*':operator.mul,
```

```
                '/':operator.truediv,
                '**':operator.pow}

        try:
            result = ops[op](a,b)
            self.res.set(result)
        except ZeroDivisionError:
            messagebox.showerror('Error', 'Zero Division Error')


def main():
    root = Tk()
    root.geometry("240x140")
    app = App(root)
    root.mainloop()


if __name__ == '__main__':
    main()
```

참고 자료

- 점프 투 파이썬 (https://wikidocs.net/book/1)
- 파이썬을 여행하는 히치하이커를 위한 안내서! (http://python-guide-kr.readthedocs.io/ko/latest/)
- Python Course (http://www.python-course.eu)