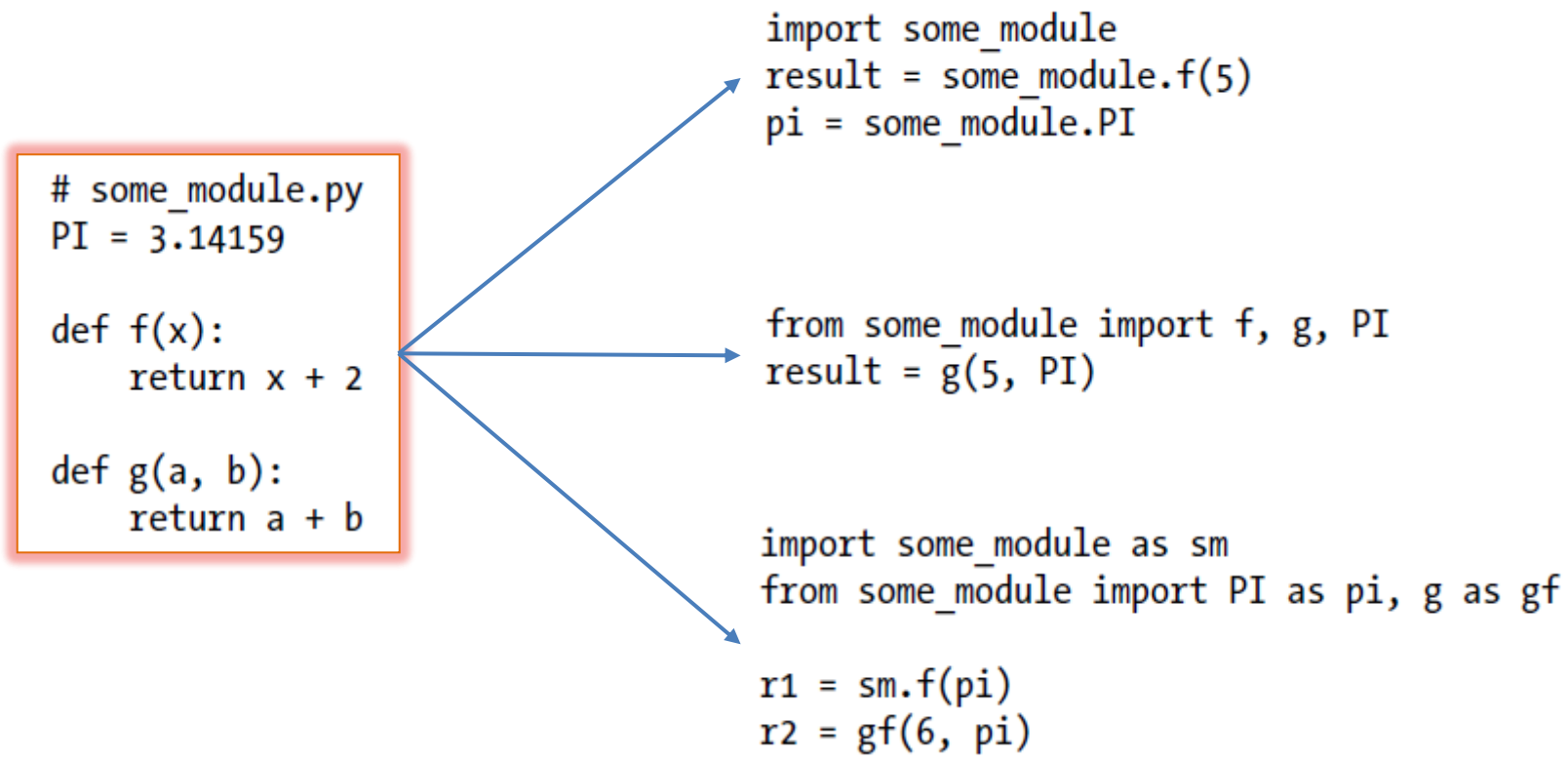
Python 5: Advanced Topics

금융공학 프로그래밍 I

Modules

- A module in Python
  - Simply a ***.py*** file containing function, class and variable definitions



Errors

1

```
In [43]: def f:

           File "<ipython-input-5-f5bdb6d29788>", line 1
             def f:
                  ^

         SyntaxError: invalid syntax
```

2

```
In [44]: 1 / 0
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-17-05c9758a9c21> in <module>()
----> 1 1/0

ZeroDivisionError: integer division or modulo by zero
```

3

```
In [45]: x1 = y1
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-23-142e0509fbd6> in <module>()
----> 1 x1 = y1

NameError: name 'y1' is not defined
```

4

```
In [47]: X = []
In [48]: x = X[0]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-22-018da6d9fc14> in <module>()
----> 1 x = X[0]

IndexError: list index out of range
```

Error Handling

```
def f(x):
    try:
        return 1.0 / x
    except ZeroDivisionError:
        print('Error: division by zero.  Returned None')
    return None
```

When we call `f` we get the following output

```
In [50]: f(2)
Out[50]: 0.5

In [51]: f(0)
         Error: division by zero.  Returned None

In [52]: f(0.0)
         Error: division by zero.  Returned None
```

Error Handling

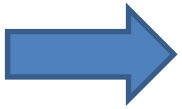
```
def f(x):
    try:
        return 1.0 / x
    except (TypeError, ZeroDivisionError):
        print('Error: Unsupported operation.  Returned None')
    return None
```

```
def f(x):
    try:
        return 1.0 / x
    except:
        print('Error.  Returned None')
    return None
```

Decorator

```python
import numpy as np

def f(x):
    assert x >= 0, "Argument must be nonnegative"
    return np.log(np.log(x))

def g(x):
    assert x >= 0, "Argument must be nonnegative"
    return np.sqrt(42 * x)
```

- Repetition of two identical lines of code



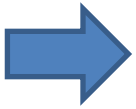
Problem?

- Look complicated
- Hard to understand

```python
import numpy as np

def check_nonneg(func):
    def safe_function(x):
        assert x >= 0, "Argument must be nonnegative"
        return func(x)
    return safe_function

def f(x):
    return np.log(np.log(x))

def g(x):
    return np.sqrt(42 * x)

f = check_nonneg(f)
g = check_nonneg(g)
```

Decorator

```
def f(x):
    return np.log(np.log(x))

def g(x):
    return np.sqrt(42 * x)

f = check_nonneg(f)
g = check_nonneg(g)
```

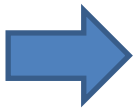


```
@check_nonneg
def f(x):
    return np.log(np.log(x))

@check_nonneg
def g(x):
    return np.sqrt(42 * x)
```

Descriptor

```
class Car(object):

    def __init__(self, miles=1000):
        self.miles = miles
        self.kms = miles * 1.61
```

Problem?



```
class Car(object):

    def __init__(self, miles=1000):
        self._miles = miles
        self._kms = miles * 1.61

    def set_miles(self, value):
        self._miles = value
        self._kms = value * 1.61

    def set_kms(self, value):
        self._kms = value
        self._miles = value / 1.61

    def get_miles(self):
        return self._miles

    def get_kms(self):
        return self._kms

    miles = property(get_miles, set_miles)
    kms = property(get_kms, set_kms)
```

Decorators and Properties

```
class Car(object):

    def __init__(self, miles=1000):
        self._miles = miles
        self._kms = miles * 1.61

    @property
    def miles(self):
        return self._miles

    @property
    def kms(self):
        return self._kms

    @miles.setter
    def miles(self, value):
        self._miles = value
        self._kms = value * 1.61

    @kms.setter
    def kms(self, value):
        self._kms = value
        self._miles = value / 1.61
```

Generator

- Generator Expressions

```
In [6]: singular = ('dog', 'cat', 'bird')

In [7]: plural = (string + 's' for string in singular)

In [8]: type(plural)
Out[8]: generator

In [9]: next(plural)
Out[9]: 'dogs'

In [10]: next(plural)
Out[10]: 'cats'

In [11]: next(plural)
Out[11]: 'birds'
```

Generator

❖ Generator Functions

```
def g(x):
    while x < 100:
        yield x
        x = x * x
```

```
In [24]: g
Out[24]: <function __main__.g>

In [25]: gen = g(2)

In [26]: type(gen)
Out[26]: generator

In [27]: next(gen)
Out[27]: 2

In [28]: next(gen)
Out[28]: 4

In [29]: next(gen)
Out[29]: 16

In [30]: next(gen)
---------------------------------------------------------
StopIteration                                       Traceb
<ipython-input-32-b2c61ce5e131> in <module>()
----> 1 gen.next()

StopIteration:
```

Everything is an Object

❖ Function

```
In [33]: def f(x): return x**2

In [34]: f
Out[34]: <function __main__.f>

In [35]: type(f)
Out[35]: function

In [36]: id(f)
Out[36]: 3074342220L

In [37]: f.__name__
Out[37]: 'f'
```

❖ Module

```
In [38]: import math

In [39]: id(math)
Out[39]: 3074329380L
```

Bracket assignment notation is just a convenient interface to a method call

```
In [30]: x = ['a', 'b']

In [31]: x.__setitem__(0, 'aa')  # Equivalent to x[0] = 'aa'
```

Namespace

- Definition
  - A symbol table that maps names to objects in memory
  - Python uses multiple namespaces
  - For example, every time we import a module, Python creates a namespace for that module

```
In [85]: import math2
In [86]: import math
In [87]: math.pi
Out[87]: 3.1415926535897931

In [88]: math2.pi
Out[88]: 'foobar'

In [90]: math.__dict__
Out[90]: {'pow': <built-in function pow>, ..., 'pi': 3.1415

In [91]: import math2

In [92]: math2.__dict__
Out[92]: {..., '__file__': 'math2.py', 'pi': 'foobar',...}
```

```
# Filename: math2.py
pi = 'foobar'
```

Namespace

- Special names
  - __doc__
  - __name__

```
In [97]: print(math.__doc__)
         This module is always available.  It provides access to the
         mathematical functions defined by the C standard.

In [98]: math.__name__
         'math'
```

- All code executed runs in some module
  - What about commands typed at the prompt?

```
In [99]: print(__name__)
         __main__
```

Global and Local Namespaces

- Global Namespace
  - The namespace of the module currently being executed

- Local Namespace
  - When we call a function, the interpreter creates a local namespace for that function
  - Local variable

- __builtins__ Namespace
  - Built-in functions: max(), dir(), str(), list(), len(), range(), type() …

```
In [12]: dir()
Out[12]: [..., '__builtins__', '__doc__', ...]  # Edited output

In [13]: dir(__builtins__)
Out[13]: [... 'iter', 'len', 'license', 'list', 'locals', ...]
```

Mutable vs. Immutable

- Immutable

```
def f(x):
    x = x + 1
    return x

x = 1
print(f(x), x)
```

- Mutable

```
def f(x):
    x[0] = x[0] + 1
    return x

x = [1]
print(f(x), x)
```

Q & A

khhwang78@gmail.com