IPA 주관 인공지능센터 기본(fundamental) 과정

- GitHub link: here
- E-Mail: windkyle7@gmail.com

함수 (Function)

Defining, calling & passing

- **parameter (매개변수)**
  - 함수 (또는 메서드) 정의에서 함수가 받을 수 있는 인자 (또는 어떤 경우 인자들) 를 지정하는 이름 붙은 엔티티. 다섯 종류의 매개변수가 있다.
    - 위치-키워드 (positional-or-keyword): `위치 인자` 나 `키워드 인자` 로 전달될 수 있는 인자를 지정한다. 이것이 기본 형태의 매개변수이다. 예를 들어 다음에서 `foo` 와 `bar` 는:

      ```
      def func(foo, bar=None):
          ...
      ```

    - 위치-전용 (positional-only): `위치로만 제공` 될 수 있는 인자를 지정한다. 파이썬은 위치-전용 매개변수를 정의하는 문법을 갖고 있지 않다. 하지만, 어떤 매장 함수들은 위치-전용 매개변수를 갖는다 (예를 들어, abs()).
    - 키워드-전용 (keyword-only): `키워드로만 제공` 될 수 있는 인자를 지정한다. 키워드-전용 매개변수는 함수 정의의 매개변수 목록에서 앞에 하나의 가변-위치 매개변수나 `*` 를 그대로 포함해서 정의할 수 있다. 예를 들어, 다음에서 `kw_only1` 와 `kw_only2` 는:

      ```
      def func(arg, *, kw_only1, kw_only2):
          ...
      ```

    - 가변-위치 (var-positional): `(다른 매개변수들에 의해서 이미 받아들여진 위치 인자들에 더해) 제공` 될 수 있는 위치 인자들의 임의의 시퀀스를 지정한다. 이런 매개변수는 매개변수 이름에 `*` 를 앞에 붙여서 정의될 수 있다. 예를 들 어 다음에서 `args` 는:

      ```
      def func(*args, **kwargs):
          ...
      ```

    - 가변-키워드 (var-keyword): `(다른 매개변수들에 의해서 이미 받아들여진 키워드 인자들에 더해) 제공` 될 수 있는 임의의 개수 키워드 인자들을 지정한다. 이런 매개변수는 매개변수 이름에 `**` 를 앞에 붙여서 정의될 수 있다. 위의 예시의 `kwargs` 가 그 예이다.
- **매개변수는 선택적 인자들을 위한 기본값뿐만 아니라 선택적이거나 필수 인자들을 지정할 수 있다.**

- Can be called both with `positional` and with `keyword` arguments
- Can be defined with `default` arguments
- Functions always return a value
  - `None` if nothing is returned explicity

```
In [1]: def roots(value):
            from math import sqrt
            result = sqrt(value)
            return result, -result
```

```
In [2]: roots(2)
Out[2]: (1.4142135623730951, -1.4142135623730951)
```

```
In [3]: def a():
            return 0
```

```
In [4]: def b():
            print(0)
```

```
In [5]: a() + 1  # Yes
        b() + 1  # No
```

```
0

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-41d3446881fd> in <module>
      1 a() + 1  # Yes
----> 2 b() + 1  # No

TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

- any object that is callable can be treated as a function
  - callable is a built-in predicate function
    - A `predicate function` is a function which purpose is to assert, such as something being either `true` or `false`.

```
In [6]: def is_at_origin(x, y):
            return x == y == 0
```

```
In [7]: is_at_origin(0, 1)
Out[7]: False
```

```
In [8]: def does_nothing():
```

```
    pass
```

In [9]:
```
assert callable(does_nothing)
```

In [10]:
```
assert does_nothing() is None
```

In [11]:
```
unsorted = ['Python', 'parrot']
```

In [12]:
```
print(sorted(unsorted, key=str.lower))  # Keyword argument
```

```
['parrot', 'Python']
```

In [13]:
```
def is_even(number):
    return number % 2 == 0
```

In [14]:
```
def print_if(values, predicate):
    for value in values:
        if predicate(value):
            print(value)
```

In [15]:
```
print_if([2, 9, 9, 7, 9, 2, 4, 5, 8], is_even)
```

```
2
2
4
8
```

- Function definitions can be nested
  - Each invocation is bound to its surrounding scope, i.e., it's a closure

In [16]:
```
def logged_execution(action, output):
    def log(message):
        print(message, action.__name__, file=output)

    log('About to execute')
    try:
        action()
        log('Successfully executed')
    except:
        log('Failed to execute')
        raise
```

In [17]:
```
world = 'Hello'
```

In [18]:
```
def outer_function():
    def nested_function():
        nonlocal world  # Refers to any world that will be assigned-
        # in within outer_function,
        # but not the global world
        world = 'Ho'

    world = 'Hi'
    print(world)
    nested_function()
    print(world)
```

In [19]:
```
outer_function()
print(world)
```

```
Hi
Ho
Hello
```

- The defaults will be substituted for corresponding missing arguments
  - Non-defaulted arguments cannot follow defaulted arguments in the definition
- Defaults evaluated once, on definition, and held within the function object
  - Avoid using mutable objects as defaults, because any changes will persist between function calls
  - Avoid referring to other parameters in the parameter list — this will either not work at all or will appear to work, but using a name from an outer scope

In [20]:
```
def line_length(x=0, y=0, z=0):
    return (x**2 + y**2 + z**2)**0.5
```

In [21]:
```
line_length()
line_length(42)
line_length(3, 4)
line_length(1, 4, 8)
```

Out[21]:
```
9.0
```

In [22]:
```
import time


def report_arg(my_default=time.time()):
    print(my_default)
```

In [23]:
```
report_arg()
time.sleep(5)
report_arg()
```

```
1557388900.8138983
```

In [24]:
```
def append_to_list(value, def_list=[]):
    def_list.append(value)
    return def_list
```

In [25]:
```
my_list = append_to_list(1)
print(my_list)
```

```
[1]
```

In [26]:
```
my_other_list = append_to_list(2)
print(my_other_list)
```

```
[1, 2]
```

- A function can be defined to take a variable argument list
  - Variadic arguments passed in as a tuple
  - Variadic parameter declared using * after any mandatory positional arguments
  - This syntax also works in assignments

In [27]:
```
def mean(value, *values):
    return sum(values, value) / (1 + len(values))
```

- To apply values from an iterable, e.g., a tuple, as arguments, unpack using *
  - The iterable is expanded to become the argument list at the point of call

In [28]:
```
def line_length(x, y, z):
    return (x**2 + y**2 + z**2)**0.5
```

In [29]:
```
point = (2, 3, 6)
length = line_length(*point)
```

In [30]:
```
point
```

Out[30]:
```
(2, 3, 6)
```

In [31]:
```
length
```

Out[31]:
```
7.0
```

- Reduce the need for chained builder calls and parameter objects
  - Keep in mind that the argument names form part of the function's public interface
  - Arguments following a variadic parameter are necessarily keyword arguments
- A function can be defined to receive arbitrary keyword arguments
  - These follow the specification of any other parameters, including variadic
  - Use ** to both specify and unpack
- Keyword arguments are passed in a dict — a keyword becomes a key
  - Except any keyword arguments that already correspond to formal parameters

In [32]:
```
def date(year, month, day):
    return year, month, day
```

In [33]:
```
sputnik_1 = date(1957, 10, 4)
sputnik_1 = date(day=4, month=10, year=1957)
```

In [34]:
```
def present(*listing, **header):
    for tag, info in header.items():
        print(tag + ': ' + info)
    for item in listing:
        print(item)
```

In [35]:
```
present('Mercury', 'Venus', 'Earth', 'Mars', type='Terrestrial', star='Sol')
```

```
type: Terrestrial
star: Sol
Mercury
Venus
Earth
Mars
```

Documentation strings

- first statement of a function, class or module can optionally be a string
  - This docstring can be accessed via **doc** on the object and is used by IDEs and the help function
  - Conventionally, one or more complete sentences enclosed in triple quotes

In [36]:
```
def echo(strings):
    """Prints space-adjoined sequence of strings."""
    print(' '.join(strings))
```

- 도큐멘테이션 문자열의 내용과 포매팅에 관한 몇 가지 관례
  - 첫 줄은 항상 객체의 목적을 짧고 간결하게 요약해야 한다. 간결함을 위해 객체의 이름이나 형을 명시적으로 언급하지 않아야 하는데, 이는 다른 방법으로 제공되기 때문이다 (이름이 함수의 작업을 설명하는 동사라면 예외). 이 줄은 대문자로 시작하고 마침표로 끝나야 한다.
  - 도큐멘테이션 문자열에 여러 줄이 있다면 두 번째 줄은 비어있기 때문에 시각적으로 요약과 나머지 설명을 분리해야 한다. 뒤따르는 줄들은 하나나 그 이

상의 문단으로, 객체의 호출 규약, 부작용 등을 설명해야 한다.
- 파이썬 파서는 여러 줄 문자열 리터럴에서 들여쓰기를 제거하지 않기 때문에 도큐멘테이션을 처리하는 도구들은 필요하면 들여쓰기를 제거한다. 이것은 다음과 같은 관례를 사용한다.
  - 문자열의 첫줄 뒤에 오는 첫 번째 비어있지 않은 줄이 전체 도튜멘테이션 문자열의 들여쓰기 수준을 결정한다. (우리는 첫줄을 사용할 수 없는데, 일반적으로 문자열을 시작하는 따옴표에 붙어있어서 들여쓰기가 문자열 리터럴의 것을 반영하지 않기 때문이다.) 이 들여쓰기와 `동등한` 공백이 문자열의 모든 줄의 시작 부분에서 제거된다. 덜 들여쓰기 된 줄이 나타나지는 말아야 하지만, 나타난다면 모든 앞부분의 공백이 제거된다. 공백의 동등성은 탭 확장 (보통 8개의 스페이스) 후에 검사된다.

Scope

- Variables are introduced in the smallest enclosing scope
  - Parameter names are local to their corresponding construct (i.e., module, function, lambda or comprehension)
- To assign to a global from within a function, declare it global in the function
  - Control-flow constructs, such as for, do not define scopes
  - del removes a name from a scope

Annotations

- annotation (어노테이션)관습에 따라 `형 힌트` 로 사용되는 변수, 클래스 어트리뷰트 또는 함수 매개변수 나 반환 값과 연결된 레이블입니다.
  - 지역 변수의 어노테이션은 실행 시간에 액세스할 수 없지만, 전역 변수, 클래스 속성 및 함수의 어노테이션은 각각 모듈, 클래스, 함수의 `__annotations__` 특수 어트리뷰트에 저장됩니다.
  - 이 기능을 설명하는 변수 어노테이션, 함수 어노테이션, PEP 484, PEP 526을 참조하세요.
- A function's parameters and its result can be annotated with expressions
  - No semantic effect, but are associated with the function object as metadata, typically for documentation purposes

In [37]:
```
def f(ham: str, eggs: str = 'eggs') -> str:
    print("Annotations:", f.__annotations__)
    print("Arguments:", ham, eggs)
    return ham + ' and ' + eggs
```

In [38]:
```
f('spam')
```

```
Annotations: {'ham': <class 'str'>, 'eggs': <class 'str'>, 'return': <class 'str'>}
Arguments: spam eggs
```

Out[38]:
```
'spam and eggs'
```

- `함수 어노테이션` 은 사용자 정의 함수가 사용하는 형들에 대한 완전히 선택적인 메타데이터 정보이다 (자세한 내용은 PEP 3107 과 PEP 484를 참고).
- 어노테이션은 함수의 `__annotations__` 어트리뷰트에 딕셔너리로 저장되고 함수의 다른 부분에는 아무런 영향을 미치지 않는다. 매개변수 어노테이션은 매개변수 이름 뒤에 오는 콜론으로 정의되는데, 값을 구할 때 어노테이션의 값을 주는 표현식이 뒤따른다. 반환 값 어노테이션은 리터럴 -> 와 그 뒤를 따르는 표현식으로 정의되는데, 매개변수 목록과 def 문의 끝을 나타내는 콜론 사이에 놓인다. 다음 예에서 위치 인자, 키워드 인자, 반환 값이 어노테이트 된다.

Decorators

- A function definition may be wrapped in decorator expressions
  - A decorator is a function that transforms the function it decorates

In [39]:
```
class List:
    @staticmethod
    def nil():
        return []

    @staticmethod
    def cons(head, tail):
        return [head] + tail
```

In [40]:
```
nil = List.nil()
```

In [41]:
```
nil
```

Out[41]:
```
[]
```

In [42]:
```
one = List.cons(1, nil)
```

In [43]:
```
one
```

Out[43]:
```
[1]
```

In [44]:
```
two = List.cons(2, one)
```

In [45]:
```
two
```

Out[45]:
```
[2, 1]
```

Lambda expressions

- Lambda calculus
- A lambda is simply an expression that can be passed around for execution
  - It can take zero or more arguments

- As it is an expression not a statement, this can limit the applicability
- Lambdas are anonymous, but can be assigned to variables
  - But defining a one-line named function is preferred over global lamba assignment

In [46]:
```
def print_if(values, predicate):
    for value in values:
        if predicate(value):
            print(value)
```

An ad hoc function that takes a single argument — in this case, number — and evaluates to a single expression — in this case, number % 2 == 0, i.e., whether the argument is even or not.

In [47]:
```
print_if([2, 9, 9, 7, 9, 2, 4, 5, 8], lambda number: number % 2 == 0)
```

```
2
2
4
8
```

Functional Programming

- first class objects = function
  - function을 first-class citizen으로 취급 (functions themselves are values or data or objects)
    - In languages where functions are not first class, functions are defined as a relationship between values, which makes them "second class".
  - 함수 자체를 인자 (argument)로써 다른 함수에 전달하거나 다른 함수의 결과값으로 반환(return)할수 있고, 함수를 변수에 할당하거나 데이터 구조 안에 저장할 수 있는 함수
    - Functions can be passed as arguments and can be used in assignment
      - This supports many callback techniques and functional programming idioms

- higher order function = fucntor
  - Takes and returns functions
    - Takes one or more functions as arguments, Returns one or more function as its result
  - Higher-order functions often used to abstract common iteration operations
    - Hides the mechanics of repetition
    - Comprehensions are often an alternative to such higher-order functions
  - Not all functions in Pythons are higher-order, because not all functions take another function as an argument. But even the functions that are not higher-order are first class. (It's a square/rectangle thing.)

- Functional Programming by Wikipidia:
  - “Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data”. In other words, functional programming promotes code with no side effects, no change of value in variables. It opposes to imperative programming, which emphasizes change of state”.
    - No mutable data (no side effect).
    - No state (no implicit, hidden state).
  - Functions are `pure` functions in the mathematical sense: their output depend only in their inputs, there is not “environment”.
    - 숨겨진 출력은 “부작용(side-effect)”으로, 숨겨진 입력은 “부원인(side-cause)”
    - 모든 입력이 입력으로 선언 (숨겨진 것이 없어야 함)
      - 마찬가지로 모든 출력이 출력으로 선언된 함수를 ‘순수(pure)’
  - Functions as principal units of `composition`
  - `Immutability`, so querying state and creating new state rather than updating it
    - Prefer to return new state rather than modifying arguments and attributes
  - Same result returned by functions called with the same inputs.
  - A `declarative` rather than imperative style, emphasizing data structure `relations`

- What are the advantages?
  - Cleaner code
    - “variables” are not modified once defined, so we don’t have to follow the change of state to comprehend what a function, a, method, a class, a whole project works.
  - Referential transparency
    - Expressions can be replaced by its values. If we call a function with the same parameters, we know for sure the output will be the same (there is no state anywhere that would change it).
  - Memoization
    - Cache results for previous function calls.
  - Idempotence
    - Same results regardless how many times you call a function.
  - Modularization
    - We have no state that pervades the whole code, so we build our project with small, black boxes that we tie together, so it promotes bottom-up programming.
  - Ease of debugging
    - Functions are isolated, they only depend on their input and their output, so they are very easy to debug.
  - Parallelization
    - Functions calls are independent.
    - We can parallelize in different process/CPUs/computers/…
    - result = func1(a, b) + func2(a, c)We can execute func1 and func2 in paralell because a won’t be modified.
  - Concurrence
    - With no shared data, concurrence gets a lot simpler:
    - No (semaphores, monitors, locks, race-conditions, dead-locks.)

FP in Python

- Functional programming tools reduce `many loops` to simple expressions
  - `Iterators` and `generators` support a lazy approach to handling series of values
    - Iterators enjoy direct protocol and control structure support in Python
    - Generators are functions that automatically create iterators
    - Generator expressions support a simple way of generating generators
    - Generators can abstract with logic
  - Declarative over imperative style, e.g., use of comprehensions over loops

- Deferred evaluation, i.e., be lazy
  - `Lazy` rather than eager access to values
  - 적용 결과 heavy computing을 줄일 수 있음.
    - 수식이 변수에 접근하는 순간 계산되는 것이 아니라, 결과를 구하려고 할 때까지 연산을 미룸으로서 불필요한 연산을 줄일 수 있는 연산 전략의 일종

- 파이썬에서, 모든 객체 메소드는 `this`를 첫번째 인자로 가진다.
- 다만 그것을 `self`라고 부를 뿐이다.

  ```
  def getName(self):
    self.name
  ```

- 분명히 명시적인 것이 묵시적인 것보다 낫다.

Functional guidance

- Immutability...
  - Prefer to return new state rather than modifying arguments and attributes
  - Resist the temptation to match every query or get method with a modifier or set
- Expressiveness...
  - Consider where loops and intermediate variables can be replaced with comprehensions and existing functions

- Resist the temptation to match every query or get method with a modifier or set
- Expressiveness...
  - Consider where loops and intermediate variables can be replaced with comprehensions and existing functions
- Python helps you write in functional style but it doesn't force you to it.
- Writing in functional style enhances your code and makes it more self documented. Actually it will make it more thread-safe also.
- The main support for FP in Python comes from the use of
  - comprehension, lambdas, closures, iterators and generators
  - modules functools and itertools.
- Python still lack an important aspect of FP: Pattern Matching and Tails Recursion.
  - There should be more work on tail recursion optimization, to encourage developers to use recursion.

Mutability pitfalls

- Python is `reference-based` language, so objects are shared by default
  - Easily accessible data or state-modifying methods can give aliasing surprises
    - An object with more than one reference has more than one name, so we say that the object is aliased.
  - **Note that true and full object immutability is not possible in Python**
- Default arguments should be of immutable type, e.g., use tuple not list
  - Changes persist between function calls

Fewer explicit loops

- A common feature of functional programming is fewer explicit loops
  - Recursion, but note that Python does not support tail recursion optimisation
  - Comprehensions — very declarative, very Pythonic
  - Higher-order functions, e.g., map applies a function over an iterable sequence
  - Existing algorithmic functions, e.g., min, str.split, str.join

Recursion

- Recursion may be a consequence of data structure traversal or algorithm
  - E.g., iterating over a tree structure
  - E.g., quicksort
- But it can be used as an alternative to looping in many simple situations
  - But beware of efficiency concerns and Python's limits (see sys.getrecursionlimit)

Reducing modifiable state

Two variables being modified explicitly <-> No variables modified

In [48]:
```
def factorial(n):
```

```
    result = 1
    while n > 0:
        result *= n
        n -= 1
    return result
```

In [49]:
```
def factorial(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

In [50]:
```
def factorial(n):
    if n > 0:
        return n * factorial(n - 1)
    else:
        return 1
```

Expressions versus statements

There is a tendency in functional programming to favour expressions...

In [51]:
```
def factorial(n):
    if n > 0:
        return n * factorial(n - 1)
    else:
        return 1
```

In [52]:
```
def factorial(n):
    return n * factorial(n - 1) if n > 0 else 1
```

In [53]:
```
factorial = lambda n: n * factorial(n - 1) if n > 0 else 1
```

But using a lambda bound to a variable instead of a single-statement function is not considered Pythonic and means factorial lacks some metadata of a function, e.g., a good `__name__`.

Imperative versus declarative

In [54]:
```
def is_leap_year(year):
    return year % 4 == 0 and year % 100 != 0 or year % 400 == 0
```

Imperative list initialisation

In [55]:
```
leap_years = []
for year in range(2000, 2030):
    if is_leap_year(year):
        leap_years.append(year)
```

In [56]:
```
leap_years
```

Out[56]:
```
[2000, 2004, 2008, 2012, 2016, 2020, 2024, 2028]
```

List comprehension

In [57]:
```
leap_years = [year for year in range(2000, 2030) if is_leap_year(year)]
```

In [58]:
```
leap_years
```

Out[58]:
```
[2000, 2004, 2008, 2012, 2016, 2020, 2024, 2028]
```

Higher-order functions

Higher-order functions - map

- map applies a function over an iterable to produce a new iterable
    - Can sometimes be replaced with a comprehension or generator expression that has no predicate
    - Often shorter if no lambdas are involved

In [59]:
```
list(map(len, 'The cat sat on the mat'.split()))
```

Out[59]:
```
[3, 3, 3, 2, 3, 3]
```

In [60]:
```
list(len(word) for word in 'The cat sat on the mat'.split())
```

Out[60]:
```
[3, 3, 3, 2, 3, 3]
```

Higher-order functions - filter

- filter includes only values that satisfy a given predicate in its generated result
    - Can sometimes be replaced with a comprehension or generator expression that has a predicate
    - Often shorter if no lambdas are involved

In [61]:
```python
scores = [100, 90, 50, 80, 50, 20, 0, 10, 100, 75]
```

In [62]:
```python
list(filter(lambda score: score > 50, scores))
```

Out[62]:
```
[100, 90, 80, 100, 75]
```

In [63]:
```python
list(score for score in scores if score > 50)
```

Out[63]:
```
[100, 90, 80, 100, 75]
```

Higher-order functions - reduce

- functools.reduce implements what is know as a fold left operation
  - Reduces a sequence of values to a single value, left to right, with the accumulated value on the left and the other on the right

In [64]:
```python
from functools import reduce
```

In [65]:
```python
def factorial(n):
    return reduce(lambda l, r: l * r, range(1, n + 1), 1)
```

`int.__mul__` would be a less general alternative:

In [66]:
```python
def factorial(n):
    return reduce(operator.mul, range(1, n + 1), 1)
```

Currying

In mathematics and computer science, currying is the techniq ue of translating the evaluation of a function that takes multi ple arguments (or a tuple of arguments) into evaluating a se quence of functions, each with a single argument (partial app lication). It was introduced by Moses Schönfinkel and later de veloped by Haskell Curry.

http://en.wikipedia.org/wiki/Currying

```
f(x, y, z) = x * y + z
```

3, 4, 5라는 인자들을 동시에 적용하여 다음과 같은 결과를 얻을 것이다.

```
f(3, 4, 5) = 3 * 4 + 5 = 17
```

하지만 3 하나만을 적용한다면, 다음과 같을 것이다.

```
f(3, y, z) = g(y, z) = 3 * y + z
```

자, 이제 두개의 인자를 가지는 g라는 새로운 함수가 생겼다.

우리는 4를 y에 적용하여 이 함수를 다시 커링할 수 있다.

```
g(4, z) = h(z) = 3 * 4 + z
```

Nested functions & lambdas

- Nested functions and lambdas bind to their surrounding scope
  - I.e., they are closures
    - 내부 함수(함수 안의 함수)가 내부 함수를 둘러싼 외부 함수의 변수나 매개변수를 이용했는데 이 외부 함수가 종료되어도 내부 함수에서 사용한 변수 값은 내부 함수 내에서 계속 유지되는 기능

In [67]:
```python
def curry(function, first):
    def curried(second):
        return function(first, second)

    return curried
```

In [68]:
```python
def curry(function, first):
    return lambda second: function(first, second)
```

In [69]:
```python
hello = curry(print, 'Hello')
hello('World')
```

```
Hello World
```

Force non-positional use of subsequent parameters:

In [70]:
```python
def timed_function(function, *, report=print):
    from time import time

    def wrapper(*args):
        start = time()
        try:
            function(*args)
        finally:
            report(time() - start)
    return wrapper
```

partial application

- Values can be bound to a function's parameters using functools.partial
  - Bound positional and keyword arguments are supplied on calling the resulting callable, other arguments are appended

In [71]:
```
from functools import partial
```

In [72]:
```
quoted = partial(print, '>')
```

In [73]:
```
quoted()   # print('>')
```

```
>
```

In [74]:
```
quoted('Hello, World!')  # print('>', 'Hello, World!')
```

```
> Hello, World!
```

In [75]:
```
quoted('Hello, World!', end=' <\n')  # print('>', 'Hello, World!', end=' <\n')
```

```
> Hello, World! <
```

Built-in algorithmic functions

```
min(iterable)
min(iterable, default=value) # Default used if iterable is empty Type
min(iterable, key=function) # The function to transform the values before determining the lowest one
max(iterable)
max(iterable, default=value)
max(iterable, key=function)
sum(iterable)
sum(iterable, start)
any(iterable)
all(iterable)
sorted(iterable)
sorted(iterable, key=function)
sorted(iterable, reverse=True)
zip(iterable, ...) # One or more iterables can be zipped together as tuples,
                   # i.e., effectively converting rows to columns,
                   # but zipping two iterables is most common
...
```

Functions from itertools

```
chain(iterable, ...)
compress(iterable, selection)
dropwhile(predicate, iterable)
takewhile(predicate, iterable)
count()
count(start)
count(start, step)
islice(iterable, stop)
islice(iterable, start, stop)
islice(iterable, start, stop, step)
cycle(iterable)
repeat(value)
repeat(value, times)
zip_longest(iterable, ...)
zip_longest(iterable, ..., fillvalue=fill)
...
```

Comprehensions

Imperative list initialisation

In [76]:
```
squares = []
for i in range(13):
    squares.append(i**2)
```

In [77]:
```
squares
```

Out[77]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144]
```

List comprehension

In [78]:
```
squares = [i**2 for i in range(13)]
```

In [79]:
```
squares
```

Out[79]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144]
```

List comprehension

In [80]:
```
[i**2 for i in range(13)]
```

Out[80]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144]
```

Set comprehension

In [81]:
```
{abs(i) for i in range(-10, 10)}
```

Out[81]:
```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Dictionary comprehension

In [82]:
```
{i: i**2 for i in range(8)}
```

Out[82]:
```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49}
```

One way to do it?

```
list(range(0, 100, 2))
list(range(100))[::2]
list(range(100)[::2])
list(map(lambda i: i * 2, range(50)))
list(filter(lambda i: i % 2 == 0, range(100)))
[i * 2 for i in range(50)]
[i for i in range(100) if i % 2 == 0]
[i for i in range(100)][::2]
```

Experiment

In what order do you expect the elements in the comprehension to appear?

In [83]:
```
values = ['A'] + list(range(2, 11)) + ['J', 'Q', 'K']
suits = ['spades', 'hearts', 'diamonds', 'clubs']
[(value, suit) for suit in suits for value in values]
```

Out[83]:
```
[('A', 'spades'),
 (2, 'spades'),
 (3, 'spades'),
 (4, 'spades'),
 (5, 'spades'),
 (6, 'spades'),
 (7, 'spades'),
 (8, 'spades'),
 (9, 'spades'),
 (10, 'spades'),
 ('J', 'spades'),
 ('Q', 'spades'),
 ('K', 'spades'),
 ('A', 'hearts'),
 (2, 'hearts'),
 (3, 'hearts'),
 (4, 'hearts'),
 (5, 'hearts'),
 (6, 'hearts'),
 (7, 'hearts'),
 (8, 'hearts'),
 (9, 'hearts'),
 (10, 'hearts'),
 ('J', 'hearts'),
 ('Q', 'hearts'),
 ('K', 'hearts'),
 ('A', 'diamonds'),
 (2, 'diamonds'),
 (3, 'diamonds'),
 (4, 'diamonds'),
 (5, 'diamonds'),
 (6, 'diamonds'),
 (7, 'diamonds'),
 (8, 'diamonds'),
 (9, 'diamonds'),
 (10, 'diamonds'),
 ('J', 'diamonds'),
 ('Q', 'diamonds'),
 ('K', 'diamonds'),
 ('A', 'clubs'),
 (2, 'clubs'),
 (3, 'clubs'),
 (4, 'clubs'),
 (5, 'clubs'),
 (6, 'clubs'),
 (7, 'clubs'),
 (8, 'clubs'),
 (9, 'clubs'),
 (10, 'clubs'),
 ('J', 'clubs'),
 ('Q', 'clubs'),
 ('K', 'clubs')]
```

iterator & generator

Being lazy

- Be lazy by taking advantage of functionality already available
  - E.g., the min, max, sum, all, any and sorted built-ins all apply to iterables
  - Many common container-based looping patterns are captured in the form of container comprehensions
  - Look at itertools and functools modules for more possibilities
- Be lazy by using abstractions that evaluate their values on demand
  - You don't need to resolve everything into a list in order to use a series of values
- Iterators and generators let you create objects that yield values in sequence
  - An iterator is an object that iterates
  - For some cases you can adapt existing iteration functionality using the iter built-in

Iteration

- An iterator is an object that iterates, following the iterator protocol
  - An iterable object can be used with for
- Iterators and generators are lazy, returning values on demand
  - You don't need to resolve everything into a list in order to use a series of values
  - Yield values in sequence, so can linearise complex traversals, e.g., tree structures
- An iterator is an object that supports the **next** method for traversal
  - Invoked via the next built-in function
- An iterable is an object that returns an iterator in support of `__iter__` method
  - Invoked via the iter built-in function
  - Iterators are iterable, so the **iter** method is an identity operation

```python
class Iterable:
...
    def __iter__(self): # An object is iterable if it supports the __iter__ special method,
                        # which is called by the iter function
        return Iterator(...)
...

    class Iterator:
        ...
        def __iter__(self): # All iterators are iterable,
                            # so the __iter__ method is an identity operation
            return self

        def __next__(self): # The __next__ special method,
                            # called by next, advances the iterator
            ...
            raise StopIteration # Iteration is terminated by raising StopIteration
            ...
    ...
```

Defining iterators

- There are many ways to provide an iterator...
  - Define a class that supports the iterator protocol directly
  - Return an iterator from another object
  - Compose an iterator with iter, using an action and a termination value
  - Define a generator function
  - Write a generator expression

Iterable

- Parallel assignment = tuple unpacking
- Parallel assignment in for loops
- Function argument unpacking = splat
- reduction functions = all, any, max, min, sum
- .sort() / sorted() 비교

iter

- Use iter to create an iterator from a callable object and a sentinel value
    - Or to create an iterator from an iterable

```
In [84]: def pop_until(stack, end):
             return iter(stack.pop, end)
```

```
In [85]: history = [1, 2, None, 4, 5]
```

```
In [86]: for popped in pop_until(history, None):
             print(popped)

         5
         4
```

```
In [87]: def repl():
             for line in iter(lambda: input('> '), 'exit'):
                 print(evaluate(line))
```

next

- Iterators can be advanced manually using next
    - Calls the `__next__` method
    - Watch out for StopIteration at the end...

```
In [88]: def repl():
             try:
                 lines = iter(lambda: input('> '), 'exit')
                 while True:
                     line = next(lines)
                     print(evaluate(line))
             except StopIteration:
                 pass
```

Generators

- A generator is a comprehension that results in an iterator object
    - It does not result in a container of values
    - Must be surrounded by parentheses unless it is the sole argument of a function

```
In [89]: (i * 2 for i in range(50))

Out[89]: <generator object <genexpr> at 0x7ff640039728>
```

```
In [90]: (i for i in range(100) if i % 2 == 0)

Out[90]: <generator object <genexpr> at 0x7ff659685150>
```

```
In [91]: sum(i * i for i in range(10))

Out[91]: 285
```

Generator expressions

- A comprehension-based expression that results in an iterator object
    - Does not result in a container of values
    - Must be surrounded by parentheses unless it is the sole argument of a function
    - May be returned as the result of a function

```
In [92]: from random import random
```

```
In [93]: numbers = (random() for _ in range(42))
```

```
In [94]: sum(numbers)

Out[94]: 18.714141684209643
```

```
In [95]: sum(random() for _ in range(42))

Out[95]: 23.19295767254536
```

Generator functions & yield

- You can write your own iterator classes or, in many cases, just use a function
    - On calling, a generator function returns an iterator and behaves like a coroutine

```
In [96]: def evens_up_to(limit):
             for i in range(0, limit, 2):
                 yield i
```

In [97]:
```
for i in evens_up_to(100):
    print(i)
```
```
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
32
34
36
38
40
42
44
46
48
50
52
54
56
58
60
62
64
66
68
70
72
74
76
78
80
82
84
86
88
90
92
94
96
98
```

Generator functions

- A generator is an ordinary function that returns an iterator as its result
  - The presence of a `yield` or `yield from` makes a function a generator, and can only be used within a function
  - `yield` returns a single value
  - `yield` from takes values from another iterator, advancing by one on each call
  - `return` in a generator raises `StopIteration`, passing it any return value specified

In [98]:
```
def medals():
    yield 'Gold'
    yield 'Silver'
    yield 'Bronze'


def medals():
    for result in 'Gold', 'Silver', 'Bronze':
        yield result


def medals():
    yield from ['Gold', 'Silver', 'Bronze']
```

In [99]:
```
for medal in medals():
    print(medal)
```
```
Gold
Silver
Bronze
```

Builtin Generation Function

- enumerate
- filter
- map
- reversed
- zip

enumerate

- Iterating a list without an index is easy... but what if you need the index?
  - Use enumerate to generate indexed pairs from any iterable

In [100]:
```
codes = ['AMS', 'LHR', 'OSL']
```

In [101]:
```
for index, code in enumerate(codes, 1):
    print(index, code)
```

```
1 AMS
2 LHR
3 OSL
```

zip

- Elements from multiple iterables can be zipped into a single sequence
  - Resulting iterator tuple-ises corresponding values together

In [102]:
```
codes = ['AMS', 'LHR', 'OSL']
names = ['Schiphol', 'Heathrow', 'Oslo']
```

In [103]:
```
for airport in zip(codes, names):
    print(airport)
```

```
('AMS', 'Schiphol')
('LHR', 'Heathrow')
('OSL', 'Oslo')
```

In [104]:
```
airports = dict(zip(codes, names))
```

In [105]:
```
airports
```

Out[105]:
```
{'AMS': 'Schiphol', 'LHR': 'Heathrow', 'OSL': 'Oslo'}
```

map

- map applies a function to iterable elements to produce a new iterable
  - The given callable object needs to take as many arguments as there are iterables
  - The mapping is carried out `on demand` and not at the point map is called

In [106]:
```
def histogram(data):
    return map(lambda size: size * '#', map(len, data))
```

In [107]:
```
text = "I'm sorry Dave, I'm afraid I can't do that."
print('\n'.join(histogram(text.split())))
```

```
###
#####
#####
###
######
#
#####
##
#####
```

filter

- filter includes only values that satisfy a given predicate in its generated result
  - If no predicate is provided — i.e., None —the Boolean of each value is assumed

In [108]:
```
numbers = [42, 0, -273.15, 0.0, 97, 23, -1]
positive = filter(lambda value: value > 0, numbers)
non_zero = filter(None, numbers)
```

In [109]:
```
list(positive)
```

Out[109]:
```
[42, 97, 23]
```

In [110]:
```
list(non_zero)
```

Out[110]:
```
[42, -273.15, 97, 23, -1]
```

- Prefer use of comprehensions over use of map and filter
  - But note that a list comprehension is fully rather than lazily evaluated

In [111]:
```
numbers = [42, 0, -273.15, 0.0, 97, 23, -1]
positive = [value for value in numbers if value > 0]
non_zero = [value for value in numbers if value]
```

```
In [112]: positive
Out[112]: [42, 97, 23]
```

```
In [113]: non_zero
Out[113]: [42, -273.15, 97, 23, -1]
```

Itertools

- infinte generators
  - count(), cycle(), repeat()
- generators that consume multiple iterables
  - chain(), tee(), izip(), imap(), product(), compress()...
- generators that filter or bundle items
  - compress(), dropwhile(), groupby(), ifilter(), islice()
- generators that rearrange items
  - product(), permutations(), combinations()

How not to iterate

```
In [114]: currencies = {
              'EUR': 'Euro',
              'GBP': 'British pound',
              'NOK': 'Norwegian krone',
          }
```

```
In [115]: for code in currencies:
              print(code, currencies[code])

          EUR Euro
          GBP British pound
          NOK Norwegian krone
```

```
In [116]: ordinals = ['first', 'second', 'third']
```

```
In [117]: for index in range(0, len(ordinals)):
              print(ordinals[index])

          first
          second
          third
```

```
In [118]: for index in range(0, len(ordinals)):
              print(index + 1, ordinals[index])

          1 first
          2 second
          3 third
```

```
In [119]: for code, name in currencies.items():
              print(code, name)

          EUR Euro
          GBP British pound
          NOK Norwegian krone
```

```
In [120]: for ordinal in ordinals:
              print(ordinal)

          first
          second
          third
```

```
In [121]: for index, ordinal in enumerate(ordinals, 1):
              print(index, ordinal)

          1 first
          2 second
          3 third
```