

Ruby 에 대해서 알아봅시다

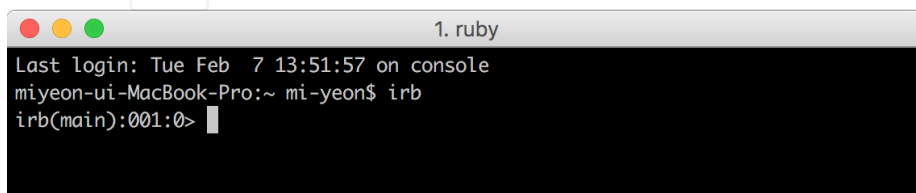
(1) irb

1. irb란 ?

입력하는 코드의 결과를 바로바로 볼수있는 프로그램

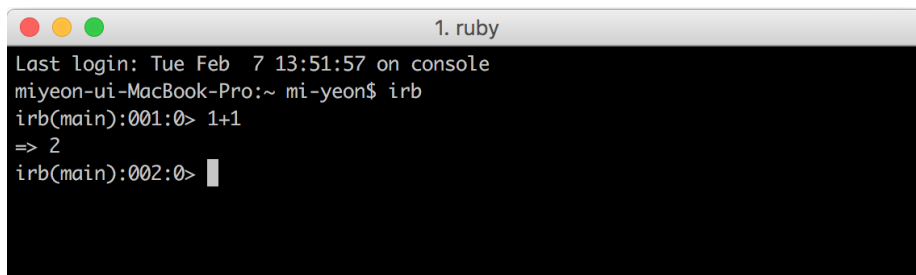
2. 사용방법

1. 터미널에 `irb` 입력

A terminal window titled '1. ruby' showing the process of starting the irb (Interactive Ruby) environment. The prompt is 'miyeon-ui-MacBook-Pro:~ mi-yeon\$'. After typing 'irb', the prompt changes to 'irb(main):001:0>'.

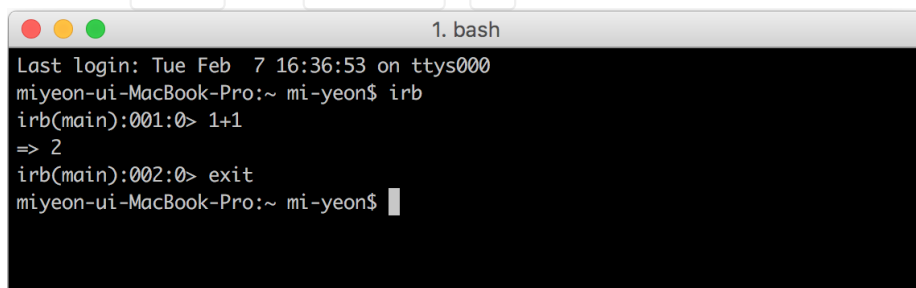
```
1. ruby
Last login: Tue Feb  7 13:51:57 on console
miyeon-ui-MacBook-Pro:~ mi-yeon$ irb
irb(main):001:0>
```

2. 코드를 입력하면 결과가 바로 출력된다.

A terminal window titled '1. ruby' showing a calculation performed in the irb environment. The prompt is 'irb(main):001:0>'. After typing '1+1', the result '=> 2' is displayed. The prompt then changes to 'irb(main):002:0>'.

```
1. ruby
Last login: Tue Feb  7 13:51:57 on console
miyeon-ui-MacBook-Pro:~ mi-yeon$ irb
irb(main):001:0> 1+1
=> 2
irb(main):002:0>
```

3. 종료할때 `exit` 또는 `control + d`

A terminal window titled '1. bash' showing the process of exiting the irb environment. The prompt is 'irb(main):002:0>'. After typing 'exit', the prompt returns to the shell 'miyeon-ui-MacBook-Pro:~ mi-yeon\$'.

```
1. bash
Last login: Tue Feb  7 16:36:53 on ttys000
miyeon-ui-MacBook-Pro:~ mi-yeon$ irb
irb(main):001:0> 1+1
=> 2
irb(main):002:0> exit
miyeon-ui-MacBook-Pro:~ mi-yeon$
```

(2) 자료형

루비에서 입력하는 데이터의 형태는 크게 세가지가 있다.

1. 숫자(numbers)

숫자를 입력 할 수 있고 수 끼리의 연산도 가능하다.

```
1. ruby
miyeon-ui-MacBook-Pro:~ mi-yeon$ irb
irb(main):001:0> 1
=> 1
irb(main):002:0> 1+3
=> 4
irb(main):003:0> 
```

2. 논리(booleans)

따옴표를 함께 사용할 경우 문자열로 인식하기 때문에 따옴표를 쓰면 안된다.

```
1. ruby
miyeon-ui-MacBook-Pro:~ mi-yeon$ irb
irb(main):001:0> true
=> true
irb(main):002:0> false
=> false
irb(main):003:0> 
```

3. 문자열(string)

대/소문자 구별도 가능하다.

```
1. ruby
miyeon-ui-MacBook-Pro:~ mi-yeon$ irb
irb(main):001:0> "likelion"
=> "likelion"
irb(main):002:0> "hello world!"
=> "hello world!"
irb(main):003:0> 
```

(3) 변수

1. 변수란?

데이터 값을 담는 데이터의 대명사

2. 변수가 필요한 이유

데이터를 한번만 쓸거라면 직접 입력해주면 되지만 재사용할 경우 변수로 저장해서 관리하는게 더 효율적이기 때문이다.

3. 저장 가능한 자료형

숫자, 논리, 문자열의 형태 모두 변수에 저장이 가능하다.

```
1. ruby
miyeon-ui-MacBook-Pro:~ mi-yeon$ irb
irb(main):001:0> my_number = 1
=> 1
irb(main):002:0> my_boolean = true
=> true
irb(main):003:0> my_string = "mi-yeon"
=> "mi-yeon"
irb(main):004:0> 
```

대입연산자를 사용하여 변수에 값 대입하기

```
my_number = 1
```

= 을 이용하여 오른쪽 의 값을 왼쪽 변수에 대입한다.

cf) 주석

코드에 부가적인 설명을 쓰거나 사용하지 않는 코드를 비활성화시키기 위해서 사용한다.

뒤에 따라오는 내용은 해석되지 않는다.

```
#한줄주석은 이렇게!
```

```
=begin
```

```
여러줄 주석은 이렇게!
```

```
=end
```

(4) 연산하기

같은 자료형 끼리는 연산이 가능하다. 또한 값을 직접 입력해 연산 할 수도 있지만 변수를 이용해서도 할 수 있다.

1. 숫자

1. 기본 숫자연산

- + 더하기
- - 빼기
- * 곱하기
- ** 제곱하기
- / 몫 구하기
- % 나머지 구하기

```
1. ruby
Last login: Tue Feb  7 16:51:25 on ttys001
miyeon-ui-MacBook-Pro:~ mi-yeon$ irb
irb(main):001:0> 1+2+4
=> 7
irb(main):002:0> 3-2
=> 1
irb(main):003:0> 5*4
=> 20
irb(main):004:0> 2**4
=> 16
irb(main):005:0> 5%3
=> 2
irb(main):006:0> █
```

2. 비교연산자

- 비교연산자는 결과값으로 boolean값이 나온다.

- A == B : 두 객체가 같은지 비교한다. A와 B가 같을 경우 true

```
miyeon-ui-MacBook-Pro:~ mi-yeon$ irb
irb(main):001:0> 1 == 1
=> true
irb(main):002:0> 1 == 3
=> false
irb(main):003:0> █
```

- A < B : 두 객체의 크기비교를 한다. A가 B보다 작을 경우 true

```
miyeon-ui-MacBook-Pro:~ mi-yeon$ irb
irb(main):001:0> 1 < 4
=> true
irb(main):002:0> 4 < 1
=> false
irb(main):003:0> █
```

- A <= B : 두 객체의 크기비교를 한다. A가 B보다 작거나 같을 경우 true

```
miyeon-ui-MacBook-Pro:~ mi-yeon$ irb
irb(main):001:0> 2 <= 3
=> true
irb(main):002:0> 3 <= 2
=> false
irb(main):003:0> █
```

- A > B: 두 객체의 크기비교를 한다. A가 B보다 클경우 true

```
miyeon-ui-MacBook-Pro:~ mi-yeon$ irb
irb(main):001:0> 4 > 3
=> true
irb(main):002:0> 3 > 4
=> false
irb(main):003:0> █
```

- A >= B: 두 객체의 크기비교를 한다. A가 B보다 크거나 같을경우 true

```
miyeon-ui-MacBook-Pro:~ mi-yeon$ irb
irb(main):001:0> 4 >= 3
=> true
irb(main):002:0> 3 >= 4
=> false
irb(main):003:0> █
```

2. 논리연산자

- && AND : 하나라도 거짓이면 거짓

```
miyeon-ui-MacBook-Pro:~ mi-yeon$ irb
irb(main):001:0> true && true
=> true
irb(main):002:0> true && false
=> false
irb(main):003:0> false && true
=> false
irb(main):004:0> false && false
=> false
irb(main):005:0> █
```

boolean1	boolean2	result

true	true	true
true	false	false
false	true	false
false	false	false

- **||** OR : 하나라도 참이면 참

```
miyeon-ui-MacBook-Pro:~ mi-yeon$ irb
irb(main):001:0> true || true
=> true
irb(main):002:0> true || false
=> true
irb(main):003:0> false || true
=> true
irb(main):004:0> false || false
=> false
irb(main):005:0> 
```

boolean1	boolean2	result
true	true	true
true	false	true
false	true	true
false	false	false

- **!** NOT : 반대로

```
miyeon-ui-MacBook-Pro:~ mi-yeon$ irb
irb(main):001:0> !true
=> false
irb(main):002:0> !false
=> true
irb(main):003:0> true!
NoMethodError: undefined method `true!' for main:Object
from (irb):3
from /Users/mi-yeon/.rbenv/versions/2.3.3/bin/irb:11:in `<main>'
irb(main):004:0> 
```

boolean1	result
!true	false
!false	true

3. 문자열

- **+** 연산

```
miyeon-ui-MacBook-Pro:~ mi-yeon$ irb
irb(main):001:0> "likelion" + "konkuk"
=> "likelionkonkuk"
irb(main):002:0> "likelionkonkuk"-"konkuk"
NoMethodError: undefined method '-' for "likelionkonkuk":String
Did you mean?  -@
               from (irb):2
               from /Users/mi-yeon/.rbenv/versions/2.3.3/bin/irb:11:in `<main>'
irb(main):003:0> █
```

- * 연산

```
miyeon-ui-MacBook-Pro:~ mi-yeon$ irb
irb(main):001:0> "likelion" * 3
=> "likelionlikelionlikelion"
irb(main):002:0> █
```

(5) 메소드

1. 메소드란?

코드의 재사용을 효율적으로 하도록 도와주는것

2. 메소드의 형식

```
def 메소드의 이름 (인자)
  실행할 코드
end
```

3. 메소드의 사용

```
var1 = 1
var2 = 2
var3 = 3

def plus_four(num)
  num += 4
  puts num
end

plus_four(var1)
```

```
plus_four(var2)
plus_four(var3)
```

다음과 같이 메소드의 이름으로 메소드를 사용하는것을 메소드를 호출한다 라고 한다.
또한 num은 인자 로, 메소드를 사용할 대상인 변수를 의미한다.

4. return

return 은 뒤에 따라오는 값을 메소드의 결과로 뱉어내며 메소드를 종료시킨다.
이때 결과를 반환한다 라고 표현한다.

```
def my_name
  return "miyeon"
end

def my_age
  return 22
end

my_name      #=> "miyeon"
my_age       #=> 22
```

5. 문자열 메소드

▶ .length

문자열의 길이를 알고싶을때 사용한다.

```
"likelion konkuk".length  #=> 15
```

▶ .reverse

문자열을 뒤집고 싶을때 사용한다.

```
"likelion".reverse  #=> noilekil
```

▶ .upcase & .downcase

문자열을 대/소문자로 바꾸고 싶을때 사용한다.

```
"likelion".upcase      #=> LIKELION
"LIKELION".downcase    #=> likelion
```

▶ .capitalize

첫 글자를 대문자로 나머지 글자는 소문자로 바꾸고 싶을때 사용한다.

```
"likelion".capitalize  #=> Likelion
```

► .include?

해당 문자를 포함하는지 아닌지를 boolean값으로 알려준다.

```
"likelion".include? "a" ==> false
"likelion".include? "i" ==> true
```

```
"likelion".include? "i"  #=> true
```

► .gsub

특정문자를 찾아 다른문자로 바꾸고 싶을때 사용한다.

```
"likelion".gsub(/i/,"a")  #=> lakelaon
```

► `.split`

문자열을 띄어쓰기 전후로 분리해서 배열로 만들어준다

```
"like lion".split ==> ["like","lion"]
```

(6) 반복문

1. 왜 반복문이 필요할까?

만약 `likelion` 을 10번 화면에 출력하고 싶다면 어떻게 할 수 있을까?

[illegible]

cf> 출력하기

1. puts

```
puts "likelion"  
puts "konkuk"
```

```
=> likelion  
     konkuk
```

2. print

```
print "likelion"  
print "konkuk"
```

```
=> likelionkonkuk
```

다음과 같이 10번을 직접 입력해 줄 수도 있겠지만 효율이 떨어진다.
이때 반복문을 사용하면 매우 간단해진다.

2. 반복문 사용방법

1. **while**: 특정조건이 `true` 인 동안 실행

```
while 조건  
  실행할 코드  
end
```

```
miyeon-ui-MacBook-Pro:~ mi-yeon$ irb  
irb(main):001:0> x = 0  
=> 0  
irb(main):002:0> while x < 5  
irb(main):003:1> puts x  
irb(main):004:1> x += 1  
irb(main):005:1> end  
0  
1  
2  
3  
4  
=> nil  
irb(main):006:0> █
```

2. until : 특정조건이 `true` 가 될 때까지 실행

until 조건

실행할 코드

end

```
miyeon-ui-MacBook-Pro:~ mi-yeon$ irb
irb(main):001:0> x = 5
=> 5
irb(main):002:0> until x < 0
irb(main):003:1> puts x
irb(main):004:1> x -=1
irb(main):005:1> end
5
4
3
2
1
0
=> nil
irb(main):006:0> 
```

3. loop : `break` 의 조건을 만족할 때까지 실행

'조건에 사용할 변수' = 0

loop do

실행할 코드

x = x + 1

break if 종료조건

end

```
miyeon-ui-MacBook-Pro:~ mi-yeon$ irb
irb(main):001:0> x = 0
=> 0
irb(main):002:0> loop do
irb(main):003:1* puts x
irb(main):004:1> x += 1
irb(main):005:1> break if x > 3
irb(main):006:1> end
0
1
2
3
=> nil
irb(main):007:0> 
```

4. for : 횟수로 반복

for x **in** 1...3

실행할 코드

end

```
miyeon-ui-MacBook-Pro:~ mi-yeon$ irb
irb(main):001:0> for x in 1...3
irb(main):002:1> puts x
irb(main):003:1> end
1
2
=> 1...3
irb(main):004:0> █
```

cf) `..` 과 `...` 의 차이

`1..10` 은 10을 포함하고

`1...10` 은 10을 포함하지 않는다.

(7) 제어문

프로그램의 실행 순서를 변경시키거나 실행할때 특정 조건이 필요한 경우 제어문을 사용한다.

1. `if` / `elsif` / `else`

조건이 `true` 일때만 해당 코드를 실행한다.

```
if 조건
  실행할 코드
elsif
  실행할 코드
else
  실행할 코드
end
```

```
miyeon-ui-MacBook-Pro:~ mi-yeon$ irb
irb(main):001:0> x = 3
=> 3
irb(main):002:0> y = 4
=> 4
irb(main):003:0> if x > y
irb(main):004:1> puts "hello lion!"
irb(main):005:1> elsif x < y
irb(main):006:1> puts "hello konkuk!"
irb(main):007:1> else
irb(main):008:1* puts "byebye!"
irb(main):009:1> end
hello konkuk!
=> nil
irb(main):010:0> █
```

2. unless / elsif / end

조건이 `false` 일때만 해당 코드를 실행한다.

unless 조건

실행할 코드

else

실행할 코드

end

```
miyeon-ui-MacBook-Pro:~ mi-yeon$ irb
irb(main):001:0> unless 3 <= 4
irb(main):002:1> puts "likelion"
irb(main):003:1> else
irb(main):004:1* puts "konkuk"
irb(main):005:1> end
konkuk
=> nil
irb(main):006:0> █
```

3. case / when / else / end

case 조건이 될 변수

when 조건

실행할 코드

else

실행할 코드

end

```
miyeon-ui-MacBook-Pro:~ mi-yeon$ irb
irb(main):001:0> my_var = "likelion"
=> "likelion"
irb(main):002:0> case my_var
irb(main):003:1> when "likelion"
irb(main):004:1> puts my_var
irb(main):005:1> when "konkuk"
irb(main):006:1> puts my_var
irb(main):007:1> else
irb(main):008:1* puts "nothing"
irb(main):009:1> end
likelion
=> nil
irb(main):010:0> █
```

(8) 배열

1. 배열이란?

데이터를 저장 할 수 있는 데이터 구조중 하나

데이터에 차례대로 부여된 숫자(index)를 통해서 데이터를 불러낼수있다.

1 2 3 4 5 6



2. 배열사용하기

```
my_array = [1, "konkuk", true]
```

다음과같이 `[]` 로 감싸주면 배열로 사용할 수 있다.

배열에는 모든 자료형이 골고루 들어갈 수 있다.

3. index

배열 내의 특정 요소를 사용하기 위해서는 **index**를 사용해야한다.

index는 `0` 부터 `배열크기-1` 까지 부여된다

```
my_array = [1, "konkuk", true]
puts my_array[0] #=> 1
puts my_array[1] #=> konkuk
puts my_array[2] #=> true
```

4. 배열 조작하기

4-1) 배열의 끝에 요소 추가하기 (<<, push)

```
arr = [1, 2, 3, 4, 5]
arr << 6
puts arr

#=> [1, 2, 3, 4, 5, 6]
```

```
arr = [1, 2, 3, 4, 5]
arr.push(6)
puts arr

#=> [1, 2, 3, 4, 5, 6]
```

4-2) 배열의 시작부분에 요소 추가하기 (unshift)

```
arr = [1, 2, 3, 4, 5]
arr.unshift(0)
puts arr

#=> [0, 1, 2, 3, 4, 5]
```

4-3) 요소 삭제하기(shift, pop)

`shift` 는 앞에서부터, `pop` 은 뒤에서부터 인자에 해당하는 수만큼 제거한다.

```
arr = [1, 2, 3, 4, 5]
arr.shift(2)
puts arr

#=> [3, 4, 5]
```

```
arr = [1, 2, 3, 4, 5]
arr.pop(2)
puts arr

#=> [1, 2, 3]
```

4-4) 순서대로 정렬하기 (sort)

```
arr = [3, 2, 1, 5, 4]
arr.sort!
puts arr
```

```
#=> [1, 2, 3, 4, 5]
```

cf) ! ?

변수에 메소드를 적용할 경우 그 변경된값은 사본에 저장된다.

하지만 ! 를 써줄 경우 변경된 값이 원본에 저장된다.

(9) 해시

1. 해시란?

데이터가 키-값쌍으로 저장되어 '키'로 데이터를 불러낼 수 있다.

2. 키-값쌍이란?

일두희 이두희 삼두희 사두희 오두희 육두희



3. 해시 사용하기

3-1) 해시 생성과 동시에 키-값도 지정해주기

```
my_hash = {
  :key1=>value1,
  :key2=>value2,
  :key3=>value3
}
```

3-2) 해시 생성 이후 키-값은 나중에 지정해주기

1번의 경우에도 이후에 아래와 같은 방법으로 키-값을 추가해줄 수 있다.

```
my_hash = Hash.new
my_hash["key1"] = value1
my_hash["key2"] = value2
```



```
my_hash["key3"] = value3
```

4. 심볼이란 ?

: + 문자열 = symbol

```
:my_sym  
:my_sym.to_s  
"likelion".to_sym
```

```
=> "my_sym"  
=> :likelion
```

- 문자열 -> 심볼 또는 심볼 -> 문자열 변환만이 가능하다.
- 한번 정해지면 값이 바뀌지 않기때문에 해쉬의 키 값으로 사용된다.

5. 키값으로 심볼 사용하기

```
new_hasg = {  
  :key1=>value1,  
  :key2=>value2,  
  :key3=>value3, }  
}
```

```
new_hash = {  
  key1: value1,  
  key2: value2,  
  key3: value3  
}
```

두가지 방법 모두 사용할 수 있다.

(10) 반복자 (Iterator)

▶ 반복자가 없다면?

```
my_arr = [1, 2, 3, 4, 5]  
my_arr[0] *= 2  
my_arr[1] *= 2  
my_arr[2] *= 2
```

```
my_arr[3] *= 2
my_arr[4] *= 2
```

다음과 같이 같은 코드를 여러번 사용해야 한다.

또한 배열의 크기가 매우 커진다면 코드는 훨씬 길어지고 따라서 실수할 가능성이 높아진다.

하지만 반복자를 사용할 경우 코드는 훨씬 간결해진다.

▶ 반복자 사용하기

```
my_arr = [1, 2, 3, 4, 5]
my_arr.each do |num|
  num *= 2
  puts num
end
```

이전의 코드와 줄 수는 차이가 별로 나지않지만 배열의 크기가 커진다면 차이는 확연히 보일것이다.

▶ 해시에서의 each

```
my_hash = {
  :key1=>value1,
  :key2=>value2,
  :key3=>value3
}

my_hash.each do |key, value|
  print key
  print value
end
```

|key, value| 에서 다른 이름의 변수를 사용하더라도 첫번째는 키, 두번째는 값이 들어간다.

▶ times : 횟수로 반복

```
반복할 횟수.times { 실행할 코드 }
```

```
miyeon-ui-MacBook-Pro:~ mi-yeon$ irb
irb(main):001:0> 3.times { puts "likelion" }
likelion
likelion
likelion
=> 3
irb(main):002:0> 
```

(11) 참고할 메소드

▶ irb에서 메소드 검색하기

`help` 를 사용하면 메소드를 검색할 수 있다.

`:q` 를 쓰면 나올수 있다.

▶ `.select`

`{}` 에 조건을 추가해서 걸러낼 수 있다.

```
grade = {
  a: 100,
  b: 90,
  c: 80,
  d: 70
}

grade.select { |name, grade| grade < 90 } #=> { :c=>80, :d=>70 }
```

▶ `.each_key`

해쉬에서 키만 뽑아낼 수 있다.

```
my_hash = {
  :key1=>1,
  :key2=>2,
  :key3=>3
}

my_hash.each_key { |key| puts key }
```

▶ `.each_value`

해쉬에서 값만 뽑아낼 수 있다.

```
my_hash = {  
  :key1=>1,  
  :key2=>2,  
  :key3=>3  
}  
  
my_hash.each_value { |value| puts value }
```

▶ `.delete`

해시의 키를 통해 키-값쌍을 제거할수있다.

```
my_hash = {  
  :key1=>1,  
  :key2=>2,  
  :key3=>3  
}  
  
my_hash.delete(:key3) #=> {:key1=>1, :key2=>2}
```

▶ `.upto` `.downto`

하나씩 더하면서 또는 빼면서 반복을 진행할수있다.

```
1.upto(10) { |i| print i, " " }    #=> 12345678910  
10.downto(1) { |i| print i, " " }  #=> 10987654321
```

▶ `.respond_to?`

메소드를 사용할 수 있는지 논리값을 반환해 알려준다.

```
[1,2,3].respond_to?(:push)    #=> true  
[1,2,3].respond_to?(:to_sym)  #=> false
```

▶ `.collect` or `.map`

`.collect` 와 `.map` 은 같은 기능이다.

배열 전체요소에 코드를 적용시킬때 사용한다.

```
a = [ "a", "b", "c", "d" ]  
a.collect { |x| x + "!" }      #=> ["a!", "b!", "c!", "d!"]  
a.map.with_index { |x, i| x * i } #=> ["", "b", "cc", "ddd"]  
a                             #=> ["a", "b", "c", "d"]
```

▶ `.floor`

해당 숫자보다 작은 정수중에서 가장 큰 정수 를 반환한다.

```
(3.14159).floor #=> 3  
(-9.1).floor    #=> -10
```

인자와 함께 넘겨줄 경우 소수자리까지 적용된다.

```
(3.14159).floor(3)      #=> 3.141  
(13345.234).floor(-2)  #=> 13300.0
```

▶ `.is_a? Object`

해당 자료형인지 확인할 수 있다.

```
:likelion.is_a? Symbol #=> true  
"like".is_a? String   #=> true  
13.is_a? Integer      #=> true
```