단원 1 - 2 장: 변수, 표현식 및 함수의 기본 개념

1 강: 변수, 표현식 및 모듈: 파이썬 프로그래밍의 핵심 기초

1. 변수(Variable)와 명령문(Statement)

이전 수업에서는 연산자를 사용하여 산술 계산을 수행하는 표현식을 작성했다.
이번 장에서는 변수와 명령문, `import` 명령문, 그리고 `print` 함수에 대해 배우게 된다.
또한 "인자(argument)"와 "모듈(module)"을 포함하여 프로그램에 대해 이야기할 때 사용하는 더 많은 어휘를 소개하겠다.

1.1. 변수(variable)

변수(variable)는 값을 참조하는 이름이다.
변수를 생성하기 위해서는 다음과 같이 **할당 명령문(assignment statement)**을 작성할 수 있다.

In [2]:
```
1 n = 17
```

할당 명령문은 세 부분으로 구성된다: 왼쪽에 있는 변수의 이름, 등호 연산자 `=` , 그리고 오른쪽에 있는 표현식이다.
이 예시에서 표현식은 정수이다.
다음 예시에서는 표현식이 부동소수점 숫자이다.

In [5]:
```
1 pi = 3.141592653589793
```

그리고 다음 예제에서 표현은 문자열이다.

In [6]:
```
1 message = 'And now for something completely different'
```

할당 명령문을 실행할 때는 출력이 없다.
파이썬은 변수를 생성하고 값을 부여하지만, 할당 명령문 자체는 눈에 보이는 효과가 없다.
하지만 변수를 생성한 후에는 이를 표현식으로 사용할 수 있다.
따라서 다음과 같이 `message` 값을 표시할 수 있다:

In [7]:
```
1 message
```

Out[7]:
```
'And now for something completely different'
```

변수를 수학 연산자와 함께 표현식의 일부로 사용할 수도 있다.

In [8]:
```
1 n + 25
```

Out[8]:
```
42
```

In [9]:
```
1 2 * pi
```

Out[9]:
```
6.283185307179586
```

함수를 호출할 때 변수를 사용할 수 있다.

In [10]:
```
round(pi)
```

Out[10]: 3

In [13]:
```
len(message)
```

Out[13]: 42

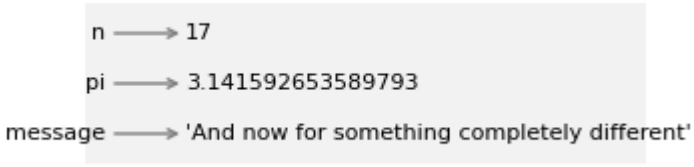
상태 다이어그램(State diagrams)

변수를 그림으로 표현하는 일반적인 방법은 이름을 쓰고 그 값을 가리키는 화살표를 그리는 것이다.

In [16]:
```
%matplotlib inline
import diagram_patch
import math

from diagram import make_binding, Frame

binding = make_binding("message", 'And now for something completely differe
binding2 = make_binding("n", 17)
binding3 = make_binding("pi", 3.141592653589793)

frame = Frame([binding2, binding3, binding])
```

diagram模块已修补，现在应该可以正常工作了。

In [17]:
```
%matplotlib inline
from diagram import diagram, adjust


width, height, x, y = [3.62, 1.01, 0.6, 0.76]
ax = diagram(width, height)
bbox = frame.draw(ax, x, y, dy=-0.25)
# adjust(x, y, bbox)
```



이러한 종류의 그림을 **상태 다이어그램(state diagram)**이라고 부름. 이는 각 변수가 어떤 상태에 있는지를 보여주기 때문이다(변수의 위치 상태라고 생각하면 됨).
이 책 전체에 걸쳐 파이썬이 변수와 그 값을 어떻게 저장하는지에 대한 모델을 표현하기 위해 상태 다이어그램을 사용할 것이다.

1.2. 변수 이름(Variable names)

변수 이름은 원하는 만큼 길게 지을 수 있다. 문자와 숫자를 모두 포함할 수 있지만, 숫자로 시작할 수는 없다.
대문자를 사용하는 것은 문법적으로 가능하지만, 변수 이름에는 관례적으로 소문자만 사용한다.
변수 이름에 사용할 수 있는 유일한 특수문자는 밑줄 문자 `_` 이다. 이는 종종 `your_name` 이나 `airspeed_of_unladen_swallow` 와 같이 여러 단어로 이루어진 이름에 사용된다. 변수에 잘못된 이름을 부여하면 구문 오류가 발생한다. `million!` 이라는 이름은 특수문자를 포함하고 있어 사용할 수 없다.

In [11]:
```
1 import thinkpython
```

In [12]:
```
1 %%expect SyntaxError
2
3 million! = 1000000
```

```
  Input In [12]
    million! = 1000000
            ^
SyntaxError: invalid syntax
```

76trombones 는 숫자로 시작하기 때문에 불법이다.

In [13]:
```
1 %%expect SyntaxError
2
3 76trombones = 'big parade'
```

```
  Input In [13]
    76trombones = 'big parade'
      ^
SyntaxError: invalid syntax
```

class 또한 불법적이지만 그 이유가 명확하지 않을 수 있다.

In [14]:
```
1 %%expect SyntaxError
2
3 class = 'Self-Defence Against Fresh Fruit'
```

```
  Input In [14]
    class = 'Self-Defence Against Fresh Fruit'
          ^
SyntaxError: invalid syntax
```

알고 보면 class 는 **키워드(keyword)**이다. 키워드는 프로그램의 구조를 지정하는 데 사용되는 특별한 단어이다.
키워드는 변수 이름으로 사용할 수 없다.
다음은 파이썬의 키워드 전체 목록이다:

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

In [1]:
```
1 from keyword import kwlist
2
3 len(kwlist)
```

Out[1]: 35

이 리스트를 외우지 않아도 되고 대부분의 개발 환경에서는, 키워드는 다른 색갈로 표시되며, 변수 이름으로 사용하려고 하면 알 수 있다.

^.^첫 번째 섹션 끝입니다. 문제를 해결하는 시간^.^

2. 표현식(Expression)과 명령문(Statement)

지금까지 우리는 몇 가지 종류의 표현식을 살펴보았다.
표현식은 정수(integer), 부동소수점 숫자(floating-point) 또는 문자열(string)과 같은 단일 값일 수 있다.
또한 값과 연산자의 집합일 수도 있다.
그리고 변수 이름과 함수 호출도 포함할 수 있다.
다음은 이러한 요소들을 여러 개 포함하는 표현식의 예이다.

In [21]:
```
1 19 + n + round(math.pi) * 2
```

Out[21]: 42

또한 몇 가지 종류의 명령문도 살펴보았다.
명령문(statement)은 효과는 있지만 값은 없는 코드 단위이다.
예를 들어, 할당 명령문은 변수를 생성하고 값을 부여하지만, 명령문 자체는 값을 가지지 않다.

In [16]:
```
1 n = 17
```

동일하게, `import` 문은 효과가 있다 -- 그것은 모듈을 가져와서 그 안에 포함된 변수와 함수를 사용할 수 있게 한다 -- 하지만 명확한 효과는 없다.

In [17]:
```
1 import math
```

표현식의 값을 계산하는 것을 **계산(evaluation)**라고 한다.
명령문을 실행하는 것을 **실행(execution)**이라고 한다.

3. import 문(명령문)

Python의 일부 기능을 사용하기 위해서는 해당 기능을 **import**(가져오기)해야 한다. 예를 들어, 다음 문장은 `math` 모듈을 가져온다.

In [1]:
```
1 import math
```

모듈(module)은 변수와 함수들의 모음이다.
math 모듈은 수학적 상수 π를 나타내는 `pi` 라는 변수를 제공한다.
다음과 같이 그 값을 표시할 수 있다.

In [18]:
```
1 math.pi
```

Out[18]:
```
3.141592653589793
```

모듈 내의 변수를 사용하려면 모듈 이름과 변수 이름 사이에 **점 연산자(dot operator)**(`.`)를 사용해야 한다.
math 모듈은 함수들도 포함하고 있다.
예를 들어, `sqrt` 는 제곱근을 계산한다.

In [19]:
```
1 math.sqrt(25)
```

Out[19]:
```
5.0
```

그리고 `pow` 는 하나의 숫자를 두 번째 숫자의 거듭제곱으로 계산한다.

In [20]:
```
1 math.pow(5, 3)
```

Out[20]:
```
125.0
```

In [22]:
```
1 5 ** 3
```

Out[22]:
```
125
```

이제까지 숫자를 제곱 연산하는 두 가지 방법을 했본다: `math.pow` 함수나 제곱 연산자 `**` 를 사용하는 2가지 방씩이 있다.
둘 중 어느 것도 괜찮지만, 연산자가 함수보다 더 자주 사용된다.

4. 출력 함수(The print function)

표현식을 계산할 때, 결과가 표시된다.

In [23]:
```
1 n = 4
2 n + 1
```

Out[23]:
```
5
```

다만, 하나 이상의 표현식을 계산할 때는 마지막 표현식의 값만 표시된다.

In [24]:
```
1 n + 2
2 n + 3
```

Out[24]:
```
7
```

하나 이상의 값을 표시하려면 `print` 함수를 사용할 수 있다.

In [25]:
```
1 print(n+2)
2 print(n+3)
```

```
6
7
```

부동소수점 숫자와 문자열과도 작동한다.

In [26]:
```
print('The value of pi is approximately')
print(math.pi)
```

```
The value of pi is approximately
3.141592653589793
```

쉼표로 구분된 표현식의 연속을 사용할 수도 있다.

In [27]:
```
print('The value of pi is approximately', math.pi)
```

```
The value of pi is approximately 3.141592653589793
```

주의: `print` 함수는 값을 사이에 공백을 두고 출력함.

5. 인수(Arguments, 實參)

함수를 호출할 때, 괄호 안에 있는 표현식을 **인수**라고 한다.

- Python 프로그래밍에서는 초보자들이 자주 혼동하는 **매개변수**와 **인수** 사이에 중요한 차이가 있다:

매개변수(Parameters, 形參)

- 함수 **정의**에 나열된 변수들
- 함수가 호출될 때 값을 받을 자리표시자 역할을 함
- 함수를 생성할 때 괄호 안에 정의됨
- "형식 매개변수(formal parameters)"라고도 불림

인수(Arguments)

- 함수가 **호출**될 때 함수에 전달되는 실제 값
- 함수 매개변수가 사용하는 데이터를 제공
- 함수를 호출할 때 괄호 안에 제공됨
- "실제 매개변수(actual parameters)"라고도 불림

간단한 예시

```
def greet(name, message):  # 'name'과 'message'는 매개변수
    print(f"{message}, {name}!")
# 함수를 호출할 때
greet("John", "Welcome")  # "John"과 "Welcome"은 인수
```

매개변수는 함수가 호출될 때 인수로 대체될 함수 정의 내의 변수라고 함. 매개변수는 함수가 필요로 하는 것을 정의하고, 인수는 함수가 받는 것을 제공한다.
이러한 구분은 함수 설계를 논의하고, 문서를 작성하며, Python뿐만 아니라 모든 프로그래밍 언어에서 코드를 디버깅할 때 중요하다.
지금까지 보았던 함수 중 일부는 `int` 와 같이 하나의 인수만 받는다.

In [ ]:
```
int('101')
```

일부는 두 개를 취한다, 예를 들어 `math.pow` .

In [ ]:
```
1 math.pow(5, 2)
```

일부는 선택 사항인 추가 인수를 받을 수 있다.
예를 들어, `int` 는 숫자의 기본을 지정하는 두 번째 인수를 받을 수 있다.

In [ ]:
```
1 int('101', 2)
```

2진수에서 `101` 숫자열은 10진수에서의 숫자 5을 나타낸다.
`round` 함수는 선택 사항으로 두 번째 인자를 받아서 반올림할 자릿수를 지정한다.

In [ ]:
```
1 round(math.pi, 5)
```

일부 함수는 어떤 수의 인수를 받을 수 있으며, `print` 와 같은 경우가 있다.

In [ ]:
```
1 print('Any', 'number', 'of', 'arguments')
```

함수를 호출할 때 인자가 너무 많이 주면 `TypeError` 가 발생한다.

In [ ]:
```
1 %%expect TypeError
2
3 float('123.0', 2)
```

충분하지 않은 인수를 제공하면 그것도 `TypeError` 이다.

In [ ]:
```
1 %%expect TypeError
2
3 math.pow(2)
```

그리고 함수가 처리할 수 없는 타입의 인자를 제공하면, 그것도 `TypeError` 이다.

In [ ]:
```
1 %%expect TypeError
2
3 math.sqrt('123')
```

이런 확인은 시작할 때 괴로울 수 있지만, 오류를 감지하고 수정하는 데 도움이 된다.

^.^두 번째 섹션 끝입니다. 문제를 해결하는 시간^.^

6. 주석(Comments)

프로그램이 커지고 복잡해질수록 읽기가 어려워진다.
형식 언어는 밀도가 높아 코드 일부를 보고 그것이 무엇을 하는지, 왜 하는지 파악하기 어려운 경우가 많다.
이러한 이유로 자연어로 프로그램이 하는 일을 설명하는 메모를 프로그램에 추가하는 것이 좋다.
이러한 메모를 **주석(comments)**이라고 하며, `#` 기호로 시작한다.

In [8]:
```
# number of seconds in 42:42
'''
asdf
sdf
sdf
asdf

'''
seconds = 42 * 60 + 42
```

이 경우, 주석은 별도의 줄에 나타난다.
또한 줄의 끝에 주석을 추가할 수도 있다:

In [ ]:
```
miles = 10 / 1.61     # 10 kilometers in miles
```

`#` 기호부터 해당 줄의 끝까지는 모두 무시된다 - 프로그램의 실행에 아무런 영향을 미치지 않다.
주석은 코드의 명확하지 않은 기능을 문서화할 때 가장 유용한다.
독자가 코드가 *무엇을(what)* 하는지 파악할 수 있다고 가정하는 것이 합리적이다; 더 유용한 것은 *왜(why)* 그렇게 하는지 설명하는 것이다.
다음 주석은 코드와 중복되며 쓸모없다:

In [ ]:
```
v = 8     # assign 8 to v
```

이 주석에는 코드에 없는 유용한 정보가 포함되어 있다:

In [ ]:
```
v = 8     # velocity in miles per hour
```

좋은 변수 이름은 주석이 필요한 경우를 줄일 수 있지만, 긴 이름은 복잡한 표현을 읽기 어려워서 균형을 맞추는 것이 필요하다.

디버깅(Debugging)

프로그램에서는 세 가지 종류의 오류가 발생할 수 있다: 구문 오류, 런타임 오류, 그리고 의미 오류.
이들을 구별하는 것은 오류를 더 빨리 추적하는 데 유용한다.

- **구문 오류(Syntax error)**: "구문"은 프로그램의 구조와 그 구조에 관한 규칙을 의미한다. 프로그램 어디에든 구문 오류가 있으면, Python은 프로그램을 실행하지 않다. 즉시 오류 메시지를 표시한다.
- **런타임 오류(Runtime error)**: 프로그램에 구문 오류가 없다면 실행이 시작될 수 있다. 하지만 무언가 잘못되면 Python은 오류 메시지를 표시하고 중지한다. 이러한 유형의 오류를 런타임 오류라고 한다. 이는 또한 예외적인 일이 발생했음을 나타내기 때문에 **예외(exception)**라고도 불림.
- **의미 오류(Semantic error)**: 세 번째 유형의 오류는 의미와 관련된 "의미적" 오류이다. 프로그램에 의미 오류가 있으면 오류 메시지를 생성하지 않고 실행되지만, 의도한 대로 작동하지 않다. 의미 오류를 식별하는 것은 프로그램의 출력을 보고 무엇을 하고 있는지 파악하려고 노력하는 역방향 작업이 필요하기 때문에 까다로울 수 있다.

아래예를 보면, 불법적인 변수 이름은 문법 오류이다.

In [ ]:
```
%%expect SyntaxError

million! = 1000000
```

타입을 지원하지 않는 연산자를 사용하면 실행 시간 오류가 발생한다.

In [ ]:
```
%%expect TypeError

'126' / 3
```

마지막으로, 의미적 오류의 예를 들어보겠다.
추정해보면 `1` 과 `3` 의 평균을 계산하고 싶었지만, 연산 순서를 잊고 이렇게 썼다:

In [ ]:
```
1 + 3 / 2
```

이 표현이 평가될 때 오류 메시지를 발생시키지 않기 때문에 문법 오류나 런타임 오류는 없다.
하지만 결과는 `1` 과 `3` 의 평균이 아니므로 프로그램은 올바르지 않다.
이는 프로그램이 실행되지만 의도한 대로 작동하지 않는 경우의 의미 오류이다.

용어집(Glossary)

변수(variable): 값을 참조하는 이름.
할당문(assignment statement): 변수에 값을 할당하는 문장.
상태 다이어그램(state diagram): 변수들의 집합과 그들이 참조하는 값들을 그래픽으로 표현한 것.
키워드(keyword): 프로그램의 구조를 지정하는 데 사용되는 특별한 단어.
import 문(import statement): 모듈 파일을 읽어 그 안에 포함된 변수와 함수를 사용할 수 있게 하는 문장.
모듈(module): 함수 정의와 때로는 다른 문장들을 포함하는 Python 코드 파일.
점 연산자(dot operator): 모듈 이름 뒤에 점과 함수 이름을 지정하여 다른 모듈의 함수에 접근하는 데 사용되는 연산자 `.`.
평가하다(evaluate): 값을 계산하기 위해 표현식의 연산을 순서대로 수행하다.
문장(statement): 명령이나 동작을 나타내는 하나 이상의 코드 줄.
실행하다(execute): 문장을 실행하고 지시된 대로 수행하다.
인수(argument): 함수가 호출될 때 함수에 제공되는 값.
주석(comment): 프로그램에 포함된 텍스트로, 프로그램에 대한 정보를 제공하지만 실행에는 영향을 미치지 않는다.
런타임 오류(runtime error): 프로그램이 오류 메시지를 표시하고 종료되게 하는 오류.
예외(exception): 프로그램이 실행되는 동안 감지되는 오류.
의미 오류(semantic error): 오류 메시지를 표시하지는 않지만 프로그램이 잘못된 일을 하게 하는 오류.

Exercises

In [ ]:
```
# This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs. Run it before working on the exercises.

%xmode Verbose
```

^.^세 번째 섹션 시작합니다. 문제를 연습하는 시간^.^

Ask a virtual assistant

다시 한번, 이 장에서 다룬 주제들에 대해 더 알아보기 위해 가상 비서를 사용하는 것을 권장한다.
제가 나열한 키워드 중 궁금한 점이 있다면, "왜 class가 키워드인가요?"나 "왜 변수 이름으로 키워드를 사용할 수 없나요?"와 같은 질문을 할 수 있다.
여러분은 `int`, `float`, `str` 이 Python 키워드가 아니라는 것을 알아챘을 수도 있다.
이들은 타입을 나타내는 변수이며, 함수로 사용될 수 있다.
따라서 이러한 이름으로 변수나 함수를 가지는 것은 *합법적*이지만, 강력히 권장되지 않다. 가상 비서에게 "왜 int, float, str을 변수 이름으로 사용하는 것이 나쁜가요?"라고 물어보세요.

또한, "Python의 내장 함수들은 무엇인가요?"라고 질문해보세요.
그 중 궁금한 것이 있다면, 더 많은 정보를 요청하세요.
이 장에서는 `math` 모듈을 가져와서 제공하는 일부 변수와 함수를 사용했다. 가상 비서에게 "math 모듈에는 어떤 변수와 함수들이 있나요?"와 "math 외에 Python의 핵심 모듈로 간주되는 것은 무엇인가요?"라고 물어보세요.

Exercise

이전 장에서 언급한 조언처럼 새로운 기능을 배울 때마다 의도적으로 오류를 만들어서 무엇이 잘못되는지 확인해 보는 것이 좋다.

- `n = 17` 이 적법하다는 것을 보았다. `17 = n` 은 어떨까요?
- `x = y = 1` 은 어떨까요?
- 일부 언어에서는 모든 문장이 세미콜론(`;`)으로 끝난다. Python 문장 끝에 세미콜론을 넣으면 어떻게 될까요?
- 문장 끝에 마침표를 넣으면 어떻게 될까요?
- 모듈 이름을 잘못 입력하고 `maath` 를 가져오려고 하면 어떻게 될까요?

Exercise

Python 인터프리터를 계산기로 사용하는 연습:

Part 1. 반지름이 r인 구의 부피는 $\frac{4}{3}\pi r^3$ 이다.
반지름이 5인 구의 부피는 얼마인가요?
`radius` 라는 변수로 시작한 다음 결과를 `volume` 이라는 변수에 할당하세요. 결과를 표시하세요. `radius` 가 센티미터 단위이고 `volume` 이 입방 센티미터 단위임을 나타내는 주석을 추가하세요.

In [ ]:
```
1 # Solution goes here
```

Part 2. 삼각함수의 법칙에 따르면 모든 x 값에 대해 $(\cos x)^2 + (\sin x)^2 = 1$이 성립한다. 이것이 42와 같은 특정 x 값에 대해 실제로 성립하는지 확인해 봅시다.
먼저 이 값으로 `x` 라는 변수를 생성한다.
그런 다음 `math.cos` 와 `math.sin` 을 사용하여 x의 사인과 코사인을 계산하고, 그 제곱의 합을 구하세요.
결과는 1에 가까워야 함. 부동소수점 연산이 정확하지 않기 때문에 정확히 1이 아닐 수 있다—이는 근사적으로만 정확하기 때문이다.

In [ ]:
```
1 # Solution goes here
```

Part 3. `pi` 외에도 `math` 모듈에 정의된 또 다른 변수는 자연 로그의 밑을 나타내는 `e` 로, 수학 표기법으로는 e라고 쓴다. 이 값에 익숙하지 않으시다면, 가상 비서에게 " `math.e` 가 무엇인가요?"라고 물어보세요. 이제 e^2를 세 가지 방법으로 계산해 봅시다:

- *`math.e`* 와 거듭제곱 연산자(`**`)를 사용한다.
- `math.pow` 를 사용하여 `math.e` 를 `2` 제곱한다.
- `math.exp` 를 사용한다. 이 함수는 인자로 값 x를 받아 e^x를 계산한다.

마지막 결과가 다른 두 결과와 약간 다르다는 것을 알 수 있을 것이다.
어떤 방법이 정확한지 알아보세요.

In [ ]:
```
1 # Solution goes here
```

In [ ]:
```
# Solution goes here
```

In [ ]:
```
# Solution goes here
```

In [ ]:
```

```

In [ ]:
```

```

2 강: 파이썬 함수, 매개변수 및 반복문: 프로그래밍의 기본 구성 요소

1. 함수

이전 수업에서 우리는 `int` 와 `float` 같은 파이썬이 제공하는 여러 함수와 `math` 모듈에서 제공하는 `sqrt` 와 `pow` 같은 몇 가지 함수를 사용했다.
이번 수업에서는 자신만의 함수를 만들고 실행하는 방법을 배우게 된다.
그리고 한 함수가 다른 함수를 호출하는 방법도 살펴볼 것이다.
예시로, 몬티 파이썬 노래(Monty Python songs)의 가사를 표시할 것이다.
이러한 재미있는 예제들은 중요한 기능을 보여준다 -- 자신만의 함수를 작성하는 능력은 프로그래밍의 기초이다.
이번 수업에서는 계산을 반복하는 데 사용되는 `for` 반복문이라는 새로운 명령문도 소개한다.

함수 정의하기(Defining new functions)

함수 정의(function definition)는 새로운 함수의 이름과 함수가 호출될 때 실행되는 명령문의 순서를 지정한다.
다음은 예시이다:

In [4]:
```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")
```

`def` 는 이것이 함수 정의임을 나타내는 키워드이다.
함수의 이름은 `print_lyrics` 이다.
변수 이름으로 사용할 수 있는 모든 것은 함수 이름으로도 사용할 수 있다.
이름 뒤의 빈 괄호는 이 함수가 어떤 인자도 받지 않는다는 것을 나타낸다.
함수 정의의 첫 번째 줄을 **헤더(header)**라고 하며 -- 나머지는 **본문(body)**이라고 한다.
헤더는 콜론으로 끝나야 하고 본문은 들여쓰기가 되어야 한다. 관례적으로 **들여쓰기는 항상 네 칸의 공백**으로 한다.
이 함수의 본문은 두 개의 print 문이다; 일반적으로 함수의 본문은 어떤 종류의 명령문이든 여러 개를 포함할 수 있다.
함수를 정의하면 **함수 객체(function object)**가 생성되며, 다음과 같이 표시할 수 있다.

In [5]:
```
print_lyrics
```

Out[5]:
```
<function __main__.print_lyrics()>
```

이 출력은 `print_lyrics` 가 인자를 받지 않는 함수임을 나타낸다.
`__main__` 은 `print_lyrics` 를 포함하는 모듈의 이름이다.
이제 함수를 정의했으므로, 내장 함수를 호출하는 것과 같은 방식으로 호출할 수 있다.

In [6]:
```
print_lyrics()
```

```
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

함수가 실행되면 본문의 문장들이 실행되며, 이는 "Timberjack Song"의 첫 두 줄을 표시한다.

2. 매개변수

우리가 본 함수 중 일부는 인자를 필요로 한다; 예를 들어, `abs` 를 호출할 때는 숫자를 인자로 전달한다.
일부 함수는 둘 이상의 인자를 받는다; 예를 들어, `math.pow` 는 밑(base)과 지수(exponent)라는 두 개의 인자를 받는다.
다음은 인자를 받는 함수의 정의이다.

In [7]:
```
def print_twice(string):
    print(string)
    print(string)
```

괄호 안의 변수 이름은 **매개변수(parameter)**이다.
함수가 호출될 때, 인자의 값이 매개변수에 할당된다.
예를 들어, 다음과 같이 `print_twice` 함수를 호출할 수 있다.

In [8]:
```
print_twice('Dennis Moore, ')
```

```
Dennis Moore,
Dennis Moore,
```

이 함수를 실행하는 것은 인자를 매개변수에 할당하고 그런 다음 함수의 본문을 실행하는 것과 동일한 효과를 낸다:

In [8]:
```
string = 'Dennis Moore, '
print(string)
print(string)
```

```
Dennis Moore,
Dennis Moore,
```

변수를 인자로 사용할 수도 있다.

In [9]:
```
line = 'Dennis Moore, '
print_twice(line)
```

```
Dennis Moore,
Dennis Moore,
```

이 예시에서 `line` 의 값이 파라미터 `string` 에 할당된다.

3. 함수 호출하기(Calling functions)

함수를 정의한 후에는 다른 함수 내에서 이 함수를 사용할 수 있다.
이를 보여주기 위해, "스팸 노래(The Spam Song)"의 가사를 출력하는 함수를 작성하겠다
(https://www.songfacts.com/lyrics/monty-python/the-spam-song
(https://www.songfacts.com/lyrics/monty-python/the-spam-song)).

```
Spam, Spam, Spam, Spam,
```

In [9]:
```
def repeat(word, n):
    print(word * n)
```

이 함수를 사용하여 노래의 첫 번째 줄을 출력할 수 있다:

In [10]:
```
spam = 'Spam, '
repeat(spam, 4)
```

```
Spam, Spam, Spam, Spam,
```

첫 두 줄을 표시하려면 `repeat` 를 사용하여 새 함수를 정의할 수 있다:

In [11]:
```
def first_two_lines():
    repeat(spam, 4)
    repeat(spam, 4)
```

그런 다음 이렇게 후출함.

In [12]:
```
first_two_lines()
```

```
Spam, Spam, Spam, Spam,
Spam, Spam, Spam, Spam,
```

마지막 세 줄을 표시하려면 다른 함수를 정의할 수 있으며, 이 함수도 `repeat` 를 사용할 수 있다:

In [13]:
```
def last_three_lines():
    repeat(spam, 2)
    print('(Lovely Spam, Wonderful Spam!)')
    repeat(spam, 2)
```

In [11]:
```
last_three_lines()
```

```
Spam, Spam,
(Lovely Spam, Wonderful Spam!)
Spam, Spam,
```

마지막으로, 한 함수를 사용하여 전체 가사를 출력할 수 있다:

In [14]:
```
def print_verse():
    first_two_lines()
    last_three_lines()
```

In [15]:
```
print_verse()
```

```
Spam, Spam, Spam, Spam,
Spam, Spam, Spam, Spam,
Spam, Spam,
(Lovely Spam, Wonderful Spam!)
Spam, Spam,
```

함수를 실행할 때 `print_verse` 는 `first_two_lines` 를 호출하고, 이는 `repeat` 를 호출하며, 이는 다시 `print` 를 호출한다.
이렇게 많은 함수들이 함께 작동하고 있다.

물론 더 적은 수의 함수로도 같은 일을 할 수 있었겠지만, 이 예제의 요점은 함수들이 어떻게 함께 작동할 수 있는지 보여주는 것이다.

^.^첫 번째 섹션 끝입니다. 문제를 해결하는 시간^.^

4. 반복(Repetition)

만약 한 절 이상을 출력하고 싶다면, `for` 문을 사용할 수 있다.
다음은 간단한 예시이다.

In [16]:
```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

In [14]:
```
for i in range(7):
    print(i)
```

```
0
1
2
3
4
5
6
```

첫 번째 줄은 콜론으로 끝나는 헤더이다.
두 번째 줄은 반드시 **들여쓰기**를 해야 하는 본문이다.
헤더는 `for` 라는 키워드로 시작하고, `i` 라는 새로운 변수와 `in` 이라는 또 다른 키워드가 있다.
여기서는 `range` 함수를 사용하여 `0` 과 `1` 두 개의 값으로 이루어진 시퀀스를 생성한다.
파이썬에서 숫자를 세기 시작할 때는 보통 `0` 부터 시작한다.
`for` 문이 실행되면, `range` 에서 첫 번째 값을 `i` 에 할당한 다음 본문의 `print` 함수를 실행하여 `0` 을 출력한다.
본문의 끝에 도달하면 다시 헤더로 돌아가는데, 이것이 이 구문이 **반복문(Repetition)**이라고 불리는 이유이다.
두 번째로 반복문을 통과할 때는 `range` 의 다음 값을 `i` 에 할당하고 이를 출력한다.
그런 다음, 그것이 `range` 의 마지막 값이기 때문에 반복문이 종료된다.
다음은 `for` 반복문을 사용하여 노래의 두 절을 출력하는 방법이다.

In [17]:
```
for i in range(2):
    print("Verse", i)
    print_verse()
    print()
```

```
Verse 0
Spam, Spam, Spam, Spam,
Spam, Spam, Spam, Spam,
Spam, Spam,
(Lovely Spam, Wonderful Spam!)
Spam, Spam,

Verse 1
Spam, Spam, Spam, Spam,
Spam, Spam, Spam, Spam,
Spam, Spam,
(Lovely Spam, Wonderful Spam!)
Spam, Spam,
```

In [15]:
```
n = 3
for i in range(n):
    print("Verse", i)
    print_verse()
    print()
```

```
Verse 0
Spam, Spam, Spam, Spam,
Spam, Spam, Spam, Spam,
Spam, Spam,
(Lovely Spam, Wonderful Spam!)
Spam, Spam,

Verse 1
Spam, Spam, Spam, Spam,
Spam, Spam, Spam, Spam,
Spam, Spam,
(Lovely Spam, Wonderful Spam!)
Spam, Spam,

Verse 2
Spam, Spam, Spam, Spam,
Spam, Spam, Spam, Spam,
Spam, Spam,
(Lovely Spam, Wonderful Spam!)
Spam, Spam,
```

함수 안에 `for` 루프를 넣을 수 있다.
예를 들어, `print_n_verses` 는 `n` 이라는 이름의 매개변수를 가지며, 이는 정수여야 하며, 주어진 수의 가사를 표시한다.

In [20]:
```
def print_n_verses(n):
    for i in range(n):
        print_verse()
        print()
```

이 예시에서는 루프 본문에서 `i` 를 사용하지 않지만, 헤더에는 반드시 변수 이름이 있어야 한다.

함수(function)를 사용하는 이유?

프로그램을 함수로 나누는 것이 왜 수고할 가치가 있는지 아직 명확하지 않을 수 있다.
여러 가지 이유가 있다:

- 새로운 함수를 만들면 명령문 그룹에 이름을 지정할 기회가 생기며, 이는 프로그램을 더 쉽게 읽고 디버깅할 수 있게 한다.
- 함수는 반복적인 코드를 제거하여 프로그램을 더 작게 만들 수 있다. 나중에 변경이 필요할 때 한 곳에서만 수정하면 된다.
- 긴 프로그램을 함수로 나누면 각 부분을 하나씩 디버깅한 다음 작동하는 전체로 조립할 수 있다.
- 잘 설계된 함수는 많은 프로그램에서 유용하게 사용된다. 한 번 작성하고 디버깅하면 재사용할 수 있다.

^.^두 번째 섹션 끝입니다. 문제를 해결하는 시간^.^

디버깅(Debugging)

디버깅은 때로는 답답할 수 있지만, 도전적이고 흥미롭고 때로는 즐거울 수도 있다.
그리고 배울 수 있는 가장 중요한 기술 중 하나이다.
어떤 면에서 디버깅은 탐정 작업과 같다.
여러분은 단서를 제공받고 여러분이 보는 결과로 이어진 사건을 추론해야 한다.
디버깅은 또한 실험 과학과도 비슷하다.
무엇이 잘못되고 있는지에 대한 아이디어가 생기면, 프로그램을 수정하고 다시 시도한다.
만약 여러분의 가설이 맞다면, 수정의 결과를 예측할 수 있고, 작동하는 프로그램에 한 걸음 더 가까워진다.
만약 여러분의 가설이 틀렸다면, 새로운 가설을 생각해내야 한다.
어떤 사람들에게는 프로그래밍과 디버깅이 같은 것이다. 즉, 프로그래밍은 원하는 대로 작동할 때까지 프로그램을 점진적으로 디버깅하는 과정이다.
그 아이디어는 작동하는 프로그램으로 시작하여 작은 수정을 가하면서 진행하면서 디버깅하는 것이다.
만약 여러분이 디버깅에 많은 시간을 쓰고 있다면, 그것은 종종 테스트를 시작하기 전에 너무 많은 코드를 작성하고 있다는 신호이다.
더 작은 단계를 밟는다면, 더 빠르게 진행할 수 있다는 것을 알게 될 것이다.

용어집(Glossary)

함수 정의(function definition): 함수를 생성하는 문장.
헤더(header): 함수 정의의 첫 번째 줄.
본문(body): 함수 정의 내부의 문장 시퀀스.
함수 객체(function object): 함수 정의에 의해 생성된 값. 함수의 이름은 함수 객체를 참조하는 변수.
매개변수(parameter): 인수로 전달된 값을 참조하기 위해 함수 내에서 사용되는 이름.
반복문(loop): 하나 이상의 문장을 실행하는 문장으로, 종종 반복적으로 실행.

Exercises

In [ ]:
```
# This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs. Run it before working on the exercises.

%xmode Verbose
```

^.^세 번째 섹션 시작합니다. 문제를 연습하는 시간^.^

Ask a virtual assistant

함수나 `for` 반복문의 문장들은 관례적으로 네 칸의 공백으로 들여쓰기를 한다.
하지만 모든 사람이 이 관례에 동의하는 것은 아닌다.
이 큰 논쟁의 역사에 대해 궁금하다면, 가상 비서에게 "파이썬에서 공백과 탭에 대해 알려줘"라고 물어보세요.
가상 비서는 작은 함수를 작성하는 데 꽤 능숙한다.

1. 여러분이 선호하는 가상 비서에게 "문자열과 정수를 입력받아 해당 문자열을 주어진 횟수만큼 출력하는 repeat라는 함수를 작성해줘"라고 요청해보세요.
2. 결과가 `for` 반복문을 사용한다면, "for 반복문 없이 할 수 있을까요?"라고 물어볼 수 있다.
3. 이 장의 다른 함수 중 하나를 골라서 가상 비서에게 작성해달라고 요청해보세요. 도전 과제는 원하는 것을 정확히 얻기 위해 함수를 정확하게 설명하는 것입니다. 이 책에서 지금까지 배운 어휘를 사용하세요. 가상 비서는 함수 디버깅에도 꽤 능숙한다.
4. 가상 비서에게 이 버전의 `print_twice` 에 무엇이 잘못되었는지 물어보세요.

```
def print_twice(string):
    print(cat)
    print(cat)
```

그리고 아래 연습 문제 중 어떤 것에 막히게 되면, 가상 비서에게 도움을 요청하는 것을 고려해보세요.

Exercise

`print_right` 라는 이름의 함수를 작성하여 문자열인 `text` 를 매개변수로 받아서 문자열을 출력하고, 출력 시 문자열의 마지막 글자가 화면의 40열에 위치하도록 충분한 개수의 앞 공백을 추가한다.

In [ ]:
```
# Solution goes here
```

힌트: `len` 함수, 문자열 연결 연산자 (+)와 문자열 반복 연산자 (*)를 사용해야 한다.
여기 예제가 어떻게 작동해야 하는지 보여주는 예제가 있다.

In [ ]:
```
print_right("Monty")
print_right("Python's")
print_right("Flying Circus")
```

Exercise

함수 `triangle` 를 작성하십시오. 이 함수는 문자열과 정수를 입력받아 주어진 높이만큼의 삼각형을 그린다. 이 삼각형은 주어진 문자열의 복사본으로 이루어진다. 다음은 레벨이 `5` 인 삼각형의 예시로, 문자열 `'L'` 을 사용한 것이다.

In [ ]:
```
# Solution goes here
```

In [ ]:
```
triangle('L', 5)
```

Exercise

사용자 정의 함수 `rectangle` 를 작성하십시오. 이 함수는 문자열과 두 개의 정수를 받아 주어진 너비와 높이로 문자열의 복사본을 사용해 사각형을 그린다. 다음은 너비 `5` 와 높이 `4` 로 `'H'` 문자열로 구성된 사각형의 예이다.

In [ ]:
```
# Solution goes here
```

In [ ]:
```
rectangle('H', 5, 4)
```

Exercise

가사 "99 병의 맥주(99 Bottles of Beer)"는 다음과 같이 시작한다:

> 99 bottles of beer on the wall
> 99 bottles of beer
> Take one down, pass it around
> 98 bottles of beer on the wall

그 다음 가사는 동일하지만 시작하는 병의 수는 98에서 97로 바뀐다. 가사는 매우 길게 계속되며, 맥주가 없을 때까지 계속된다.

`bottle_verse` 라는 함수를 작성하십시오. 이 함수는 매개변수로 숫자를 받아 해당 숫자의 병이 있는 가사를 표시한다.

힌트: 각 가사의 첫 번째, 두 번째 또는 마지막 줄을 출력할 수 있는 함수를 먼저 작성한 다음, 이를 사용하여 `bottle_verse` 를 작성하십시오.

In [ ]:
```
# Solution goes here
```

In [ ]:
```
# Solution goes here
```

이 함수 호출을 사용하여 첫 번째 음절을 표시하세요.

In [ ]:
```
bottle_verse(99)
```

전부 노래를 인쇄하고 싶다면, 이 `for` 루프를 사용할 수 있다. 이 루프는 `99` 부터 `1` 까지 카운트다운한다.

이 예제를 완전히 이해할 필요는 없다---나중에 `for` 루프와 `range` 함수에 대해 더 배울 것이다.

In [ ]:
```
for n in range(99, 0, -1):
    bottle_verse(n)
    print()
```

In [ ]:
```

```