Chapter 2

변수와 수식 그리고 문장

프로그래밍 언어의 가장 강력한 기능 중 하나는 **변수(variable)**가 조작 가능하다는 것이다. 변수는 값에 부여하는 이름이다.

2.1 할당문

할당문(assignment statement)은 새로운 변수를 생성하고 값을 부여한다.

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.141592653589793
```

이 예제에서는 세 개의 변수에 값을 할당했다. 첫 번째는 `message`라는 이름의 변수에 문자열을 할당하였다. 두 번째는 `n`에 정수 `17`을 부여하였고 세 번째에는 π의 값을 `pi`라는 변수에 할당하였다.

종이에 변수를 표현하는 방법은 이름과 값을 적고 화살표를 이름에서 값으로 향하게 그리면 된다. 이와 같은 그림을 **상태도(state diagram)**이라 부른다. 각 변수의 상태를 나타내기 때문이다 (변수 마음 상태로 생각해보자) 그림 2.1는 방금의 예제의 상태도를 보여준다.

2.2 변수명

일반적으로 프로그래머는 변수에 의미있는 이름을 부여하며, 그 변수가 무엇에 이용되는지 문서화하기도 한다.

이름은 길게 만들고 싶은만큼 길게 만들 수 있다. 글자와 숫자로 이루어질 수 있지만 숫자로 시작될 수는 없다. 대문자를 못 쓰는 것은 아니지만 관례적으로 변수명으로는 소문자만을 쓴다.

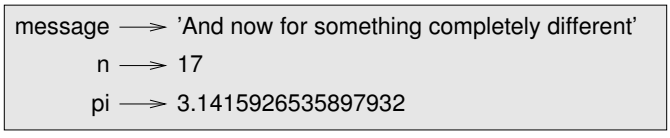

Figure 2.1: 상태도.

밑 줄 표시를, _, 이름에서 볼 수도 있다 대체적으로 your_name이나 airspeed_of_unladen_swallow처럼 여러 단어로 이루어진 이름에서 사용된다.

무효한 이름을 변수에 할당하게 되면 문법 오류 메시지가 뜬다:

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

76trombones가 잘못된 이유는 숫자로 시작하기 때문이고 more@에서 오류가 발생한 이유는 변수명으로 쓸 수 없는 글자 @를 사용했기 때문이다. 그렇다면 class에는 무슨 문제가 있는 걸까?

class는 Python이 정의해 놓은 **키워드(keyword)**이기 때문이다. 인터프리터는 키워드들을 사용하여 프로그램의 구조를 파악하기 때문에 변수명으로 사용할 수 없다.

Python 3에는 다음과 같은 키워드들이 있다:

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

이 목록을 외울 필요는 없다. 대부분의 개발 환경은 키워드들을 다른 색으로 표현해 준다. 이 목록에 있는 키워드 중 하나를 변수 명으 사용하려 한다면 알 수 밖에 없다.

2.3 수식과 문장

수식(expression)은 값과 변수 그리고 연산자들의 조합이다. 값으로만 이루어진 그 자체도 수식으로 인식되며 변수도 마찬가지이다. 다음의 유효한 수식들을 살펴보자.

```
>>> 42
42
>>> n
17
>>> n + 25
42
```

프롬프트에 수식을 입력하면 인터프리터는 수식을 **계산(evaluate)**하여 그 안에 있는 값을 찾는다. 이 예제에서는 n은 17이라는 값을 갖고 있고 n + 25 는 42라는 값을 갖고 있다.

문장(statement)은 변수를 생성한다거나 값을 표시하는 것과 같이 결과가 있는 코드의 단위이다.

```
>>> n = 17
>>> print(n)
```

첫 줄은 할당문으로 n에 값을 부여하고 있고 두 번째 줄은 n에 할당된 값을 출력하는 print문이다.

문장을 입력하면, 인터프리터는 **실행(excute)**한다. 실행한다는 의미는 그 문장이 뜻하는 데로 동작한다는 것이다. 일반적으로 문장에는 값이 할당되지 않는다.

2.4 스크립트 모드

지금까지 Python을 **대화식 모드(interactive mode)**로 인터프리터에 입력한 결과를 즉시 확인할 수 있었다. 대화식 모드는 감을 잡고 시작하기에는 유용하지만 여러 줄의 코드를 사용하기에는 불편하다.

그 대안으로 **스크립트(script)**라고 부르는 파일에 코드를 저장해 놓고 인터프리터를 **스크립트 모드(script mode)**로 동작시켜서 스크립트를 실행시키는 것이다. 통상적으로 Python 스크립트 파일의 이름은 `.py`로 끝이 난다.

컴퓨터에서 스크립트를 생성하고 실행시키는 방법을 안되면 이제 제대로 시작할 준비가 되었다. 그게 아니라면 `PythonAnywere`를 다시 한 번 사용하기를 권한다. 스크립트 모드를 실행하는 방법을 `http://tinyurl.com/thinkpython2e`에 기록해 놓았다.

Python이 두 가지 모드를 지원하기 때문에 스크립트로 저장하기 전에 대화식 모드로 실험해볼 수 있다. 하지만 대화식 모드와 스크립트 모드간의 차이가 있기 때문에 그렇게 사용하면 혼란스러울 수 있다.

예를 들어 Python을 계산기로 사용하여 다음과 같이 입력하였다고 해보자.

```
>>> miles = 26.2
>>> miles * 1.61
42.182
```

첫 줄은 `miles`에 값을 할당하였지만, 그 자체로는 가시적인 결과가 보이지 않는다. 두 번째 줄은 수식이기 때문에 인터프리터가 계산하여 결과를 표시한다. 이 결과를 보면 마라톤의 길이는 42 킬로미터 정도된다.

이 똑같은 코드를 스크립트로 저장하여 실행시켜 보면 어떤 결과도 출력되지 않는다. 스크립트 모드에서의 수식 그 자체로는 어떠한 가시적인 결과가 나타나지 않는다. 사실 Python은 수식을 계산하지만 값을 출력하라는 말을 듣기 전까지는 그 결과를 표시하지 않는다.

```
miles = 26.2
print(miles * 1.61)
```

이 같은 동작이 처음에는 이해가 안될 수 있다.

스크립트는 대체적을 연속된 문장들로 구성되어 있다. 하나 이상의 문장이 있는 경우에 각 문장이 실행될 때마다 하나씩 결과를 나타낸다.

예를 들어, 다음 같은 스크립를 살펴보자.

```
print(1)
x = 2
print(x)
```

이 스크립트를 실행하면 다음의 결과를 얻는다.

```
1
2
```

할당문은 결과를 표시하지 않는다.

이해를 했는지 확인해보기 위해 다음의 문장들을 Python 인터프리터에 입력하여 어떻게 동작하는지 살펴보라.

```
5
x = 5
x + 1
```

그리고 이 문장들을 스크립트로 저장하여 실행해보자. 결과가 어떻게 되는가? 스크립트의 각 수식을 `print`문으로 변환하여 다시 실행해보자.

2.5 연산의 우선순위

어떤 수식에 하나 이상의 연산자가 있다면 연산의 순서는 **연산의 우선순위(order of operations)**에 의존한다. 수학 연산자의 경우 Python은 일반적인 수학 연산자의 우선순위를 따른다. 한 가지 방법으로 **PEMDAS**라는 두문문자로 우선순위를 기억하면 쉽다.

- **P**arentheses(괄호)는 가장 높은 우선순위를 갖고 있기 때문에 괄호를 사용하면 원하는 데로 연산의 순서를 조작할 수가 있다. 수식에서 괄호는 가장 먼저 계산되기 때문에 `2 * (3-1)`은 4가 되고 `(1+1)**(5-2)`는 8이 된다. 또한 `(minute * 100) / 60`라는 식처럼 괄호를 사용하여 계산 결과는 바꾸지 않으면서 수식을 좀 더 읽기 쉽게 만들 수도 있다.

- **E**xponentiation(거듭제곱)이 그 다음으로 높은 우선 순위를 갖고 있다. `1 + 2**3`의 결과는 9이지 27이 아니며, `2 * 3**2`의 결과는 36이 아니라 18이다.

- **M**ultiplication(곱셈)과 **D**ivision(나눗셈)은 **A**ddition(덧셈)과 **S**ubtraction(뺄셈)보다 더 높은 우선순위를 갖고 있다. `2*3-1`의 연산 결과는 4가 아니라 5이며 `6+4/2`의 결과는 8이지 5가 아니다.

- 같은 연산 우선순위를 갖는 연산자는 왼쪽부터 오른쪽으로(거듭제곱은 예외) 계산된다. `degrees / 2 * pi`라는 식에서는 나눗셈이 먼저 계산되고 그 결과에 `pi`가 곱해진다. 만약 2π로 나누기 원한다면 괄호를 쓰거나 `degrees / 2 / pi`로 식을 고치면 된다.

일부러 연산 우선순위를 외우려고 엄청나게 노력할 필요는 없다. 수식이 바로 파악이 안된다면 괄호를 써서 그 식이 당연해지도록 바꾸면 되기 때문이다.

2.6 문자열 연산

일반적으로 문자열에 수학적 연산을 쓸 수 없다. 설령 문자열이 숫자처럼 보일지라도 그렇게 쓰는 것은 불법이다.

```
'2'-'1'    'eggs'/'easy'    'third'*'a charm'
```

그 규칙에는 두 가지 예외가 있다. `+`와 `*`이다.

`+` 연산자는 **문자열 연결(string concatenation)**할 때 사용하는 것으로 문자열들을 앞뒤로 붙여준다. 예를 들어 보자:

```
>>> first = 'throat'
>>> second = 'warbler'
>>> first + second
throatwarbler
```

`*` 연산자도 역시 문자열을 다룰 때 쓸 수 있다. `*`는 반복할 때 사용된다. 예를 들어 `'Spam'*3`의 결과는 `'SpamSpamSpam'`이다. 이것을 사용할 때는 한 값이 문자열이어야 하고 다른 값은 정수여야 한다.

`+`와 `*`의 문자열에 대한 용법은 수식에서의 덧셈과 곱셈의 용법과 연결지어 생각하면 좋다. `4*3`은 `4+4+4`와 동치이기 때문에 `'Spam'*3`의 결과로 `'Spam'+'Spam'+'Spam'`과 같은 결과를 기대할 것이고, 실제로 같은 결과를 갖고 있다. 물론, 문자열 연결과 반복은 정수의 덧셈과 곱셈과 다른 면이 많이 있다. 정수의 덧셈의 성질 중 문자열 연결의 성질과 다른 것을 생각해볼 수 있겠는가?

2.7 주석

프로그램의 크기가 커지고 복잡해지게되면 읽기가 어려워진다. 형식 언어는 난해하다고 말을 했었다. 그렇기 때문에 코드의 일부분을 보고 무슨 일을 하는지 또는 왜 그 일을 하는지 이해하지 못할 때가 많다.

그렇기 때문에 프로그램에 자연어로 이 프로그램이 어떤 일을 하는지 기록해 놓는 것은 언제나 좋은 생각이다. 이렇게 기록하는 것을 보고 **주석(comment)**라고 부른다. 주석은 # 기호로 시작한다.

```
# 한 시간에 몇 퍼센트가 지났는지 계산
percentage = (minute * 100) / 60
```

이 경우에는 주석이 혼자 한 줄을 다 쓰고 있다. 주석을 그 줄의 끝에 적을 수도 있다.

```
percentage = (minute * 100) / 60      # 시간의 퍼센트
```

인터프리터는 # 이후부터 그 줄의 *끝*까지의 모든 내용을 무시한다. 프로그램 실행에 전혀 영향을 주지 않는다.

주석이 가장 빛을 발하는 순간은 코드 중에 불명확한 기능을 설명할 때이다. 코드를 읽었을 때 그 프로그램이 *무엇*을 하는지는 알아 낼 수 있을 것이다. 그렇기 때문에 *왜* 그 일을 하는지를 설명하는 것이 가장 유익하다.

다음의 주석은 코드가 하는 일에 대한 중복 설명이라 불필요하다.

```
v = 5     # assign 5 to v
```

반면 다음의 코드는 코드에서 알아 낼 수 없는 유익한 정보를 담고 있다.

```
v = 5     # meters/second 단위의 속도
```

변수 명을 잘 정하면 주석을 달아야 할 필요가 없어지지만 너무 긴 이름으로 지으면 읽기가 어려워지기 때문에 적당한 지점에서 타협을 해야 한다.

2.8 디버깅

프로그램에는 문법(syntax) 오류, 실행시간(runtime) 오류, 그리고 문맥(semantic) 오류라는 세 가지 종류의 오류가 있을 수 있다. 이 셋을 구분할 줄 아는 것은 그 오류를 추적하는데 큰 도움을 준다.

문법(Syntax) 오류: "문법(Syntax)"은 프로그램의 구조와 그 구조에 대한 규칙을 뜻한다. 예를 들어 괄호는 두 개가 한 쌍으로 사용되어야 하기 때문에 `(1+2)`는 유효하지만, `8)`는 **문법 오류**로 사용할 수 없다.

만약 프로그램 어딘가에 문법 오류가 있다면 Python은 오류 메시지를 출력하고 종료하기 때문에 그 이후의 프로그램은 더 실행 할 수 없다. 프로그래밍에 입문한지 얼마지나지 않은 몇 주 동안은 문법 오류를 해결하는데 대부분의 시간을 활애할 것이다. 좀 더 연륜이 쌓인다면 오류의 수가 더 적어 질 것이고 그런 오류를 더 빨리 찾아 낼 수 있게 된다.

실행시간(Runtime) 오류: 두 번째 오류는 실행시간 오류라고 불리는데, 그렇게 불리는 이유는 프로그램이 실행되기 전까지는 있는지 알 수 없기 때문이다. 이러한 오류들은 **예외(exception)**이라고 불리기도 한다. 예외적인(그리고 나쁜) 일이 발생했다는 것을 나타내기 때문이다.

실행시간 오류는 이 책의 처음 몇 장을 다루는 동안에는 보게 되는 간단한 프로그램에서는 드물게 나타난다. 실제 이런 류의 오류를 만나려면 시간이 좀 흘러야 할 것이다.

문맥(Semantic) 오류: 세 번째 종류의 오류는 "문맥(semantic)" 오류로서 사용된 문장들의 의미와 관련이 있다. 프로그램에 문맥 오류가 있다면 오류 메시지 없이 실행이 잘 되지만 그렇다고 제대로 동작하는 것은 아니다. 엉뚱하게 동작을 한다. 정확하게 말하자면, 프로그래머가 시킨 일을 그대로 한다.

문맥 오류를 찾아내는 것은 까다로운 일인 이유는 프로그램이 어디서 어떻게 잘못되었는지를 프로그램의 실행 결과를 통해 역 추적해야하기 때문이다.

2.9 용어 해설

변수(variable): 값을 가리키는 이름.

할당(assignment): 변수에 값을 지정하는 문장.

상태도(state diagram): 변수와 그 변수가 가리키는 값을 표현한 그림

키워드(keyword): 프로그래밍 언어에서 미리 예약해 놓은 단어로서 프로그램을 구문해석하는 과정에 사용된다. `if`, `def`와 `while`과 같은 키워드는 변수 명으로 사용할 수 없다.

피연산자(operand): 연산자를 사용하여 계산할 때 사용되는 값

수식(expression): 변수와 연산자 그리고 값들의 조합으로 하나의 결과를 나타냄.

계산(evaluate): 하나의 값을 얻기 위해 연산하여 수식을 간단화하는 것

문장(statement): 명령이나 어떤 동작을 뜻하는 코드의 일부분. 지금까지는 봐왔던 문장은 할당하는 것과 `print`문이 있다.

실행(execute): 문장에 적힌 그대로를 수행

대화식 모드(interactive mode): 프롬프트에 코드를 입력하여 Python 인터프리터를 사용하는 방식

스크립트 모드(script mode): 스크린트 파일에 저장되어 있는 코드를 읽어 실행하도록 Python 인터프리터를 사용하는 방식

스크립트(script): 파일에 저장되어 있는 프로그램

연산의 우선순위(order of operations): 여러 개의 연산자와 피연산자로 이루어진 수식이 계산되는 순서를 결정하는 규칙

연결(concatenate): 두 개의 피 연산자를 앞뒤로 합치는 것

주석(comment): 다른 프로그래머(또는 소스 코드를 읽는 누군가)를 위해 프로그램에 기록되어 있는 정보. 프로그램의 실행에는 아무런 영향이 없음.

문법 오류(syntax error): 프로그램 해석이 불가능하게 만드는 프로그램 내의 오류(구문해석이 불가능함)

예외(exception): 프로그램이 실행 중에 발견되는 오류

문맥(semantics): 프로그램의 의미

문맥 오류(semantic error): 프로그램의 의도와 다르게 프로그램이 동작하게 만드는 오류

2.10 연습 문제

문제 2.1. *앞 장에서 했던 조언처럼 새로운 기능을 배우면 대화식 모드에서 실험해보고 의도적으로 실수를 해서 무엇이 잘못되는가를 살펴봐야 한다.*

- `n = 42`*는 유효하다는 것을 보았다.* `42 = n`*은 어떨까?*
- `x = y = 1` *은 유효한가?*
- *어떤 프로그래밍 언어는 세미콜론(*`;`*)으로 문장이 끝난다. Python의 문장의 끝을 세미콜론으로 끝을 내면 어떻게 되는가?*
- *문장의 끝을 구두점으로 끝을 내면 어떻게 되는가?*
- *수학에서는 x와 y의 곱을 표기 할 때 xy로 써도 된다. Python에서 이렇게 쓰면 어떻게 될까?*

문제 2.2. *Python의 인터프리터를 계산기처럼 사용하여 다음 문제를 실습해보자.*

1. *반지름이 r인 구의 부피는* $\frac{4}{3}\pi r^3$ *이다. 이때 구의 반지름이 5라면 부피가 어떻게 되는가?*
2. *책의 가격이 27,500원으로 책정되어 있는데, 책 방에서 40% 도서할인전을 하고 있다. 한 권일 때는 배송비가 2,500원인데, 추가 구입을 할 때마다 150원이 추가된다. 60권을 산다면 배송비를 포함한 총 금액이 얼마인가?*
3. *집에서 6시52분에 나와서 1마일을 천천히 8분 15초에 달리고, 그 다음 3마일은 마일당 7분 12초가 걸렸다. 그리고 1마일은 처음처럼 천천히 달렸다. 운동을 마치고 집에 도착하면 몇 시에 아침 식사를 할 수 있을까?*