Python statements: assignments, flow control

Intro. to Computing

`https://mandarm.github.io/computing.html`

Assignments

- Simple assignments: `a = b + 2*c`
- Chains: `a = b = c = 0`
- Multiple assignments: `y, z, r = 9.2, -7.6, 0`
- Swapping two variables: `a, b = b, a`
- Augmented assignments: `+=` `-=` `*=` `/=` `//=` `%=`

Warning: multiple assignments

- If the same variable appears on the left **AND** right sides of an assignment, right side refers to the current/existing value, left side refers to the new value (see swapping above).
- If the same variable appears **multiple times** in the LHS, the occurrences are processed **left-to-right**.

```
x = [0, 1]
i = 0
i, x[i] = 1, 2 # i is updated, THEN x[i] is updated
print(x)
```

Control flow: if statements

```
if ⟨condition1⟩:     # NOTE the colon (:) at the end
    ......
    ......
elif ⟨condition 2⟩: # NOTE the colon (:) at the end
    ......
    ......
else:                # NOTE the colon (:) at the end
    ......
    ......
```

- *List of statements to be executed for any condition is determined by indentation.*
- Zero or more `elif` parts permitted
- `else` part optional

Control flow: match statement

```
match ⟨expression⟩
    case ⟨value1⟩:
        ......
    case ⟨value2⟩:
        ......
    case ⟨value3⟩ | ⟨value4⟩:
        # Do this if expression is value3 or value4
        ......
    case _: # What to do if none of the above matches
        ......
```

- Statement(s) for only the first matching pattern get(s) executed.
- Underscore (_) serves as a “catch all” that matches everything (this is optional).

Later: more complicated matching

Control flow: for and while loops

```
for ⟨variable⟩ in ⟨container⟩:
    ......

for ⟨variable⟩ in ⟨iterator⟩:
    ......
```

Avoid code that modifies a collection while iterating over that same collection.

```
while ⟨condition⟩:
    ......
```

Control flow: `break`, `continue`

- `break`: jump out of the innermost enclosing `for` or `while` loop
- `continue`: skip the rest of the body and start the next iteration of the innermost enclosing loop

Example:

```
for divisor in range(2, x-1): # x must be an integer >= 4
    if x % divisor == 0:
        print(x, "is composite")
        break
    if divisor * divisor > x:
        print(x, "is prime")
        break
```

Addendum: `pass` statement ≡ do nothing

```
if ⟨condition⟩:
   pass    # Too complicated, will write later
else:
   ......  # This part is easier, so wrote it now
```

Iterables: `range()`, `enumerate()` I

- `range(n)`
- `range(start, stop, step)`: `step` optional, **default is 1**
- `enumerate(l)`

Example:

```
list(range(5, 10))            # [5, 6, 7, 8, 9]
list(range(0, 10, 3))         # [0, 3, 6, 9]
list(range(-10, -100, -30))   # [-10, -40, -70]

l = ['Mary', 'had', 'a', 'little', 'lamb']
for i in range(len(l)):
    print(i, l[i])
for i, w in enumerate(l):
    print(i, w)
```

Iterables: `range()`, `enumerate()` II

Iterables

- Do not create a complete list
- Generate successive items of the desired sequence **on demand** until nothing more can be generated
- `for` statement generates items of the desired sequence and acts on them **one by one**

Iterables: `range()`, `enumerate()` III

Example:

```
range(4)          # range(0, 4)
list(range(4))    #  0 + 1 + 2 + 3 = 6
sum(range(4))     #  0 + 1 + 2 + 3 = 6

list(range(20,-10,-2))
# [20, 18, 16, 14, 12, 10, 8, 6, 4, 2, 0, -2, -4, -6, -8]

sorted(range(20,-10,-2))
# [-8, -6, -4, -2, 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

# BUT!!!
reversed(range(10))        # <range_iterator object at ...>
reversed(list(range(10))) # <list_reverseiterator object at ...>
```