CSGE601020 | Dasar-Dasar Pemrograman 1

CONTROL

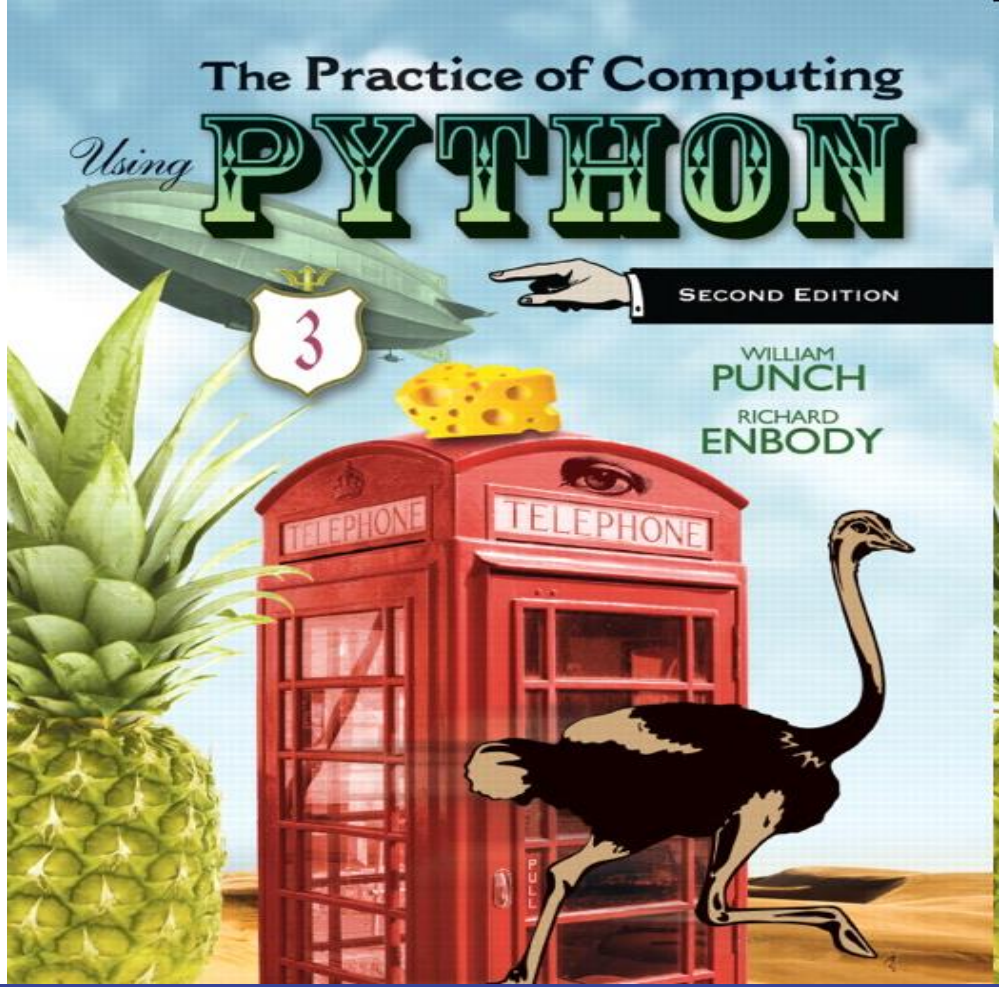
Semester Gasal 2022/2023

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia
FAKULTAS ILMU KOMPUTER

References

The Practice of Computing Using Python, Third Edition

Chapter 2



Outline

- Selection
- Repetition

Selection

Selection

Selection is how programs make **choices (decisions)**

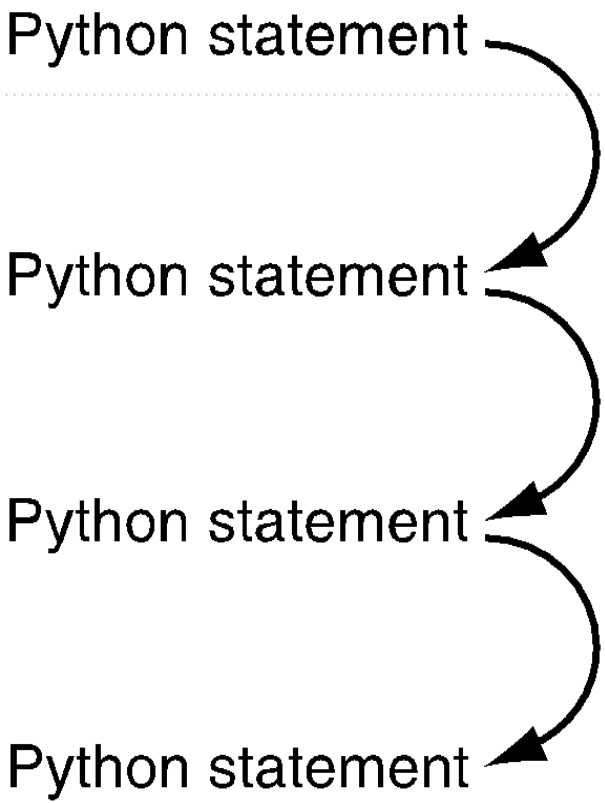


FIGURE 2.1 Sequential program flow.

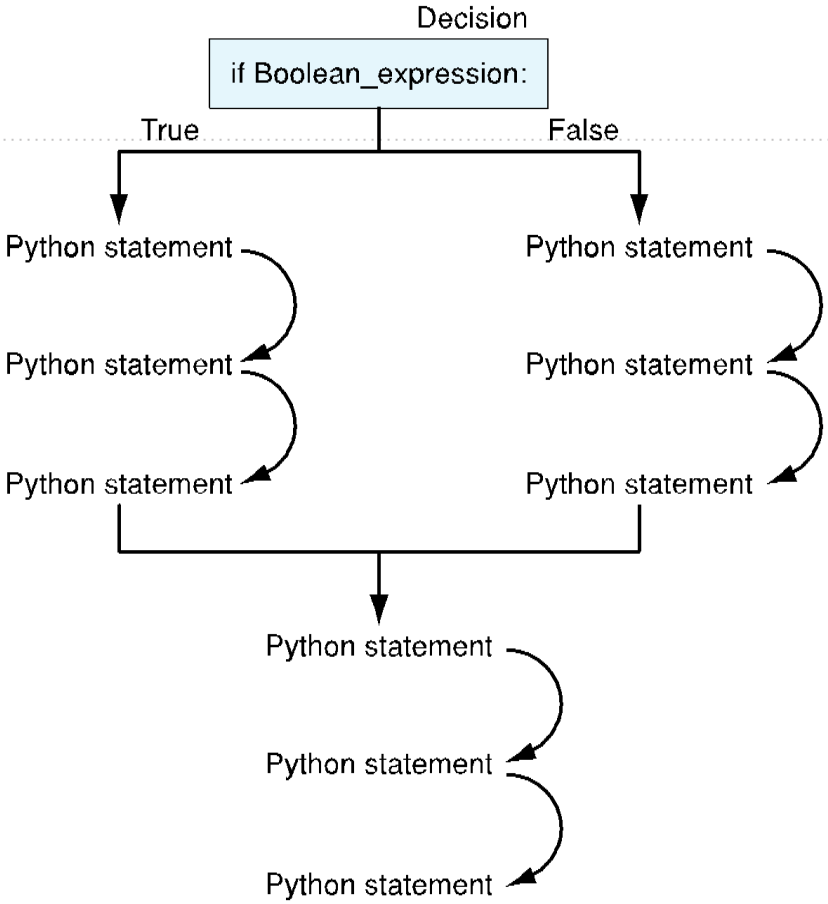


FIGURE 2.2 Decision making flow of control.

Boolean Expressions

- Boolean expressions: an expression that is true or false
  - x greater than 5
  - answer == ‘YES’
- Boolean operators: return True/False
- Every boolean expression has the form:
  - `expression booleanOperator expression`
- The result of evaluating something like the above is also just true or false.

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
!=	not equal to

TABLE 2.1 Boolean Operators.

Note that **==** is equality, **=** is assignment

Python selection, Round 1

```
if boolean expression :
      suite
```

- evaluate the boolean (`True` or `False`)
- if `True`, execute all statements in the suite

Indentation

compound statement

```
keyword expression :
        statement
        statement
        statement
```

header

'suite' of statements

Indentation under the keyword

Indentation has dual purposes:

- to indicate compound statements (indicate the structure of the code)
- to make compound statements easier to read

Warning about indentation

- Elements of the suite must all be indented the same number of spaces/tabs
- Python only recognizes suites when
  they are indented the same distance (***standard is 4 spaces***)
- You must be careful to get the indentation right to get suites right.

Python Selection, Round 2

```
if boolean expression:
      suite1
else:
      suite2
```

The process is:

- evaluate the Boolean expression
- if `True`, run suite1
- if `False`, run suite2

```
>>> first_int = 10
>>> second_int = 20
>>> if first_int > second_int:
        print("The first int is bigger!")
    else:
        print("The second int is bigger!")


The second int is bigger!
>>>
```

Live Coding: Apakah nilai input lebih dari 1000?

```
import turtle

s = turtle.Screen()
masukan = s.numinput("Contoh Program", "Masukkan angka > 1000:")

if(masukan > 1000):
    turtle.write("YES :)")
else:
    turtle.write("NO :((")

turtle.exitonclick()
```

Booleans

Boolean Expressions

- **George Boole**'s (mid-1800's) mathematics of logical expressions
- Boolean expressions (conditions) have a value of True or False
- Conditions are the basis of choices in a computer, and, hence, are the basis of the appearance of intelligence in them.

What is True, and what is False

- true: **any nonzero number or nonempty object**.
  `1, 100, "hello", [a,b]`
- false: **a zero number or empty object**.
  `0, "",[ ]`
- Special values called `True` and `False`, which are just substitutions for 1 and 0. However, they print nicely (`True` or `False`)

Relational Operators

- `Subset of Boolean operator.` *Relational* operators can be used to compare the relation between two values.
- `3 > 2` ⇒ True
- Relational Operators have low preference
  - `5 + 3 < 3 - 2`
  - `8 < 1` ⇒ False
- `'1' < 2` ⇒ Error
  - can only compare values of the same types
- `int('1') < 2` ⇒ True
  - same types, regular comparison

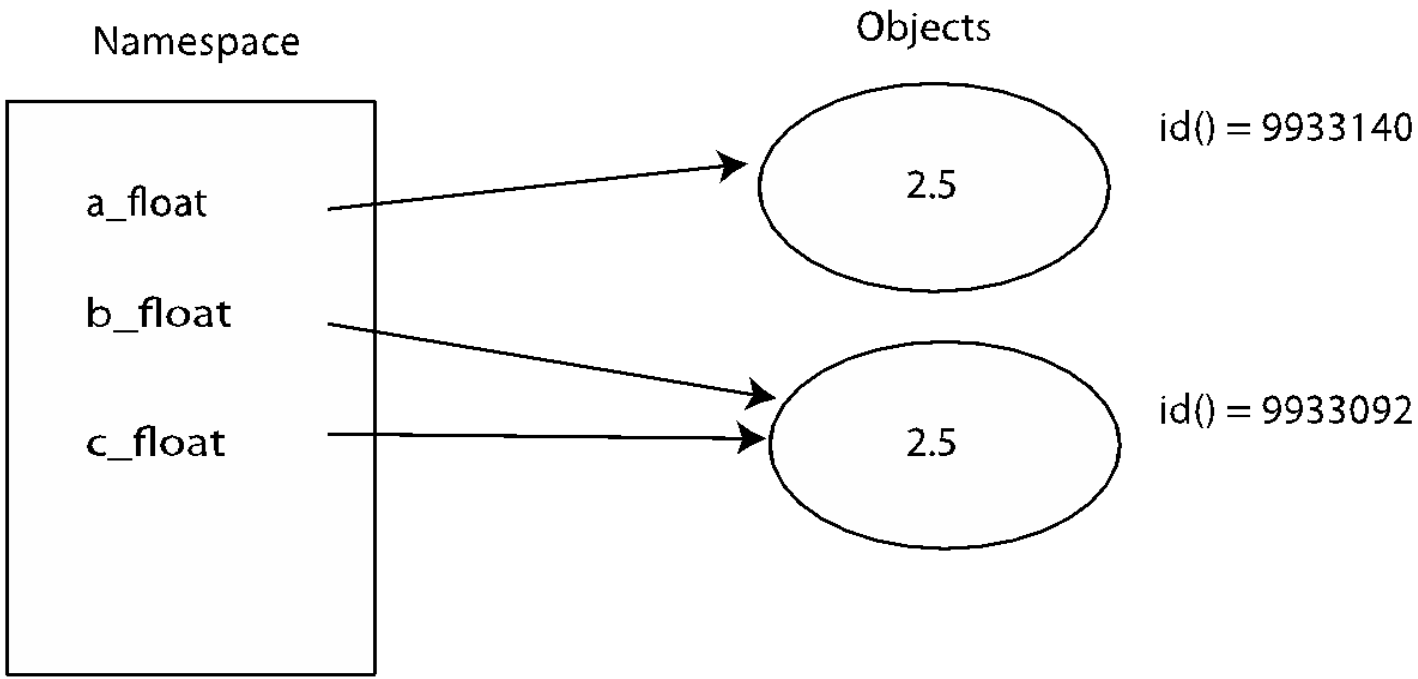
What does Equality mean?

Two senses of equality:

- two variables refer to different objects, each object representing the same value
- two variables refer to the same object. The `id()` function is used for this.

a_float = 2.5
b_float = 2.5
c_float = b_float

FIGURE 2.6 What is equality?



equal vs. same

- == compares values between the objects of two variables
- is operator determines if two variables are associated with the ***same object***

```
>>> a_int = 5
>>> b_int = 5
>>> a_int == b_int
True
>>> a_int is b_int
True
>>>
>>>
>>> a_float = 5.0
>>> b_float = 5.0
>>> a_float == b_float
True
>>> a_float is b_float
False
```

Floating point arithmetics

```
>>> u = 11111113
>>> v = -11111111
>>> w = 7.51111111
>>> (u + v) + w == u + (v + w)
```

Pitfall

- floating point arithmetic is **approximation**!

```
>>> u = 11111113
>>> v = -11111111
>>> w = 7.51111111
>>> (u + v) + w
9.51111111
>>> u + (v + w)
9.511111110448837
>>> (u + v) + w == u + (v + w)
False
```

compare using "close enough"

- Establish a level of "close enough" for equality

```
>>> u = 11111113
>>> v = -11111111
>>> w = 7.51111111
>>> x = (u + v) + w
>>> y = u + (v + w)
>>> x == y
False
>>> abs(x - y) < 0.0000001  # abs is absolute value
True
```

Chained comparisons

- In Python, chained comparisons work just like you would expect in a mathematical expression:
- Given `myInt` has the value 5
  - `0 <= myInt <= 5` → `True`
  - `0 < myInt <= 5 < 1` → `False`

Compound Expressions

Python allows bracketing of a value between two Booleans, as in math

```
a_int = 5
0 <= a_int <= 10  ⇒  True
```

- `a_int >= 0` and `a_int <= 10`
- **`and`, `or`, `not`** are the three Boolean operators in Python

Truth Tables

p	q	not p	p and q	p or q
True	True	False	True	True
True	False	False	False	True
False	True	True	False	True
False	False	True	False	False

Compound Evaluation

- Logically `0 < a_int < 3` is actually `(0 < a_int) and (a_int < 3)`
- Evaluate using `a_int` with a value of 5: `(0< a_int) and (a_int < 3)`
- Parenthesis first: `(True) and (False)`
- Final value: `False`

- (Note: parenthesis are not necessary in this case.)

Precedence & Associativity

Operator	*Description*
()	Parenthesis (grouping)
**	Exponentiation
+x, -x	Positive, Negative
*,/,%	Multiplication, Division, Remainder
+,-	Addition, Subtraction
<, <=, >, >=,! =, ==	Comparisons
not x	Boolean NOT
and	Boolean AND
or	Boolean OR

TABLE 2.2 Precedence of Relational and Arithmetic Operators: Highest to Lowest

Boolean operators (and, or) vs. relational operators

- Relational operations always return `True` or `False`
- Boolean operators (**`and, or`**) are different in that:
  - They can return values (that represent `True` or `False`)
  - They have ***short circuiting***

Short circuiting in Boolean operators

OPERATION	RESULT	NOTES
X or Y	If X is False, then Y, else X	Y is executed only if X is False Else if X is true, X is result.
X and Y	If X is false, then X else Y	Y is executed only if X is true, else if X is false , X is result.
not X	If X is true, then false, else true	not has lower priority than non - boolean operators. Eg. not a==b => not (a==b)

```
>>> 1 or 5
1
>>> 0 or 1
1
>>> 0 and 1
0
>>> 1 and 5
5
>>> not(2)
False
>>> not(0)
True
>>> x = 0 or 5
>>> x
5
>>> y = 1 and 5
>>> y
5
>>> z = not(0)
>>> z
True
```

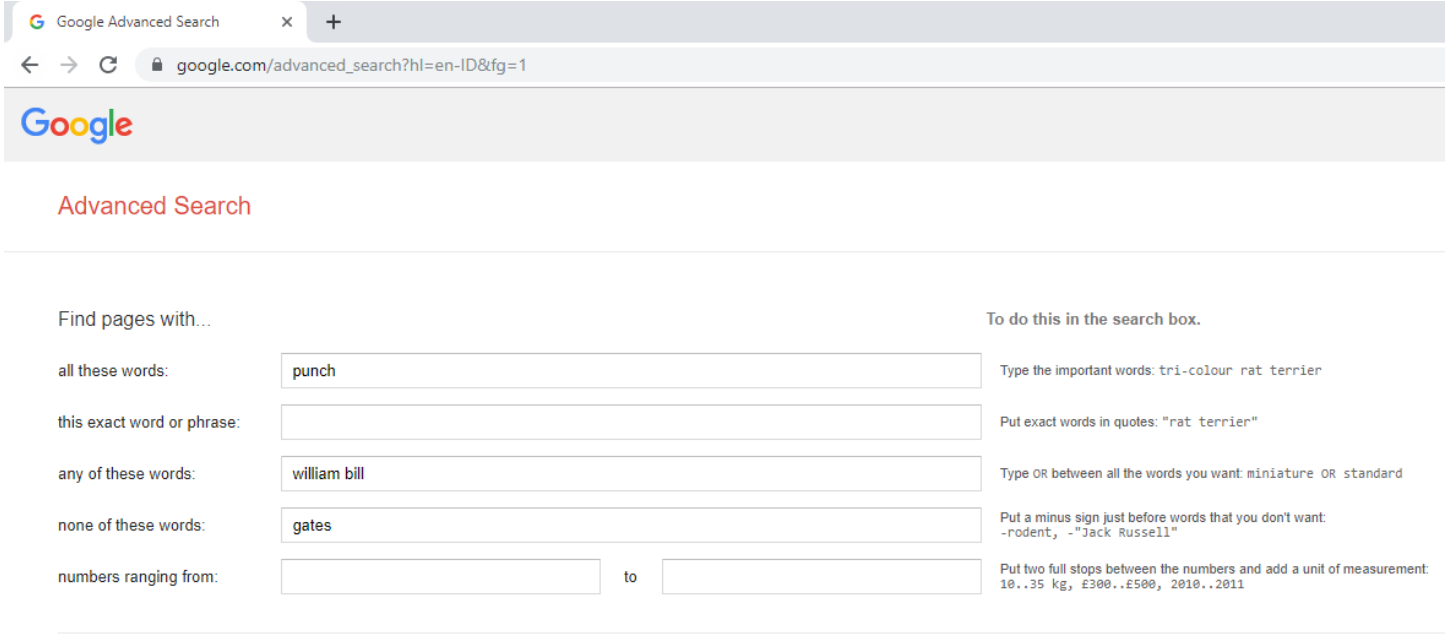
References: https://www.geeksforgeeks.org/short-circuiting-techniques-python/

Remember!

- `0, '',[ ]` or other “empty” objects are equivalent to `False`
- anything else is equivalent to `True`

Advanced Search on Google

- Example: `'Punch' and ('Bill' or 'William') and not 'gates'`



Live Coding: Apakah x > 1000 dan y < 10?

```
import turtle

s = turtle.Screen()
x = s.numinput("Contoh Program", "Masukkan angka > 1000:")
y = s.numinput("Contoh Program", "Masukkan angka < 10:")

if(x > 1000) and (y < 10):
    turtle.write("YES :)")
else:
    turtle.write("NO!")

turtle.exitonclick()
```

More on Assignments

Remember Assignments?

- Format: `lhs = rhs`
- Behavior:
  - expression in the **rhs** is evaluated producing a value
  - the value produced is placed in the location indicated on the **lhs**

Can do multiple assignments

```
a_int, b_int = 2, 3
```

first on right assigned to first on left, second on right assigned to second on left

```
print(a_int, b_int)   # prints 2 3
```

```
a_int,b_int = 1,2,3 ⍰  Error
```

counts on lhs and rhs must match

traditional swap

- Initial values: `a_int= 2, b_int = 3`
- Behavior: swap values of `X` and `Y`
  - introduce extra variable `temp`
    - `temp = a_int` # save a_int value in temp
    - `a_int = b_int` # assign a_int value to b_int
    - `b_int = temp` # assign temp value to b_int

Swap using multiple assignment

- `a_int, b_int = 2, 3`
- `print(a_int, b_int)` # prints 2 3

- `a_int, b_int = b_int, a_int`
- `print(a_int, b_int)` # prints 3 2

- remember, evaluate all the values on the **rhs** first, then assign to variables on the **lhs**

Chaining for assignment

Unlike other operations which chain left to right, assignment chains right to left

```
a_int = b_int = 5
print(a_int, b_int)  # prints 5  5
```

Compound Statements

- Compound statements involve a set of statements being used as a group
- Most compound statements have:
  - a header, ending with a `:` (colon)
  - a suite of statements to be executed
- **`if, for, while`** are examples of compound statements

We have seen 2 forms of selection

```
1). if boolean-expression:
        suite

2). if boolean-expression:
        suite
    else:
        suite
```

Python Selection, Round 3

```
if boolean-expression1:
      suite1
elif boolean-expression2:
      suite2
as many elif's as you want)
else:
      suite_last
```

`if, elif, else`, the process

- evaluate Boolean expressions until:
  - the Boolean expression returns `True`
  - none of the Boolean expressions return `True`
- if a boolean returns `True`, run the corresponding suite. Skip the rest of the `if`
- if no boolean returns `True`, run the `else` suite, the default suite

Live Coding: Nilai DDP 1 Hard Mode (Oops!)

```
import turtle

s = turtle.Screen()
nilai = s.numinput("Contoh Program", "Masukkan nilai:")

if nilai <= 100:
    turtle.write("A")
elif nilai < 90:
    turtle.write("B")
else:
    turtle.write("E")

turtle.exitonclick()
```

Live Coding: Nilai DDP 1 Hard Mode

```
import turtle

s = turtle.Screen()
nilai = s.numinput("Contoh Program", "Masukkan nilai:")

if 90 <= nilai <= 100:
    turtle.write("A")
elif 80 <= nilai < 90:
    turtle.write("B")
else:
    turtle.write("E")

turtle.exitonclick()
```

Live Coding: Nilai DDP 1 Hard Mode

```
import turtle

s = turtle.Screen()
nilai = s.numinput("Contoh Program", "Masukkan nilai:")

if 90 <= nilai <= 100:
    turtle.write("A")
elif 80 <= nilai < 90:
    turtle.write("B")
else:
    turtle.write("E")

turtle.exitonclick()
```

What happens if **elif** is replaced by **if** ?

Repetition

Repeating statements

- Besides selecting which statements to execute, a fundamental need in a program is repetition
  - repeat a set of statements under some conditions
- With both selection and repetition,
  we have the two most necessary programming ingredients

`while` and `for` statements

- The **`while`** statement is more general. It repeats a set of statements while some condition is True.
- The **`for`** statement is useful for iteration, moving through all the elements of data structure, one at a time.

`while` loop

- Top-tested loop (pretest):
  - test the boolean before each iteration of the loop (*incl. the first iteration*)

```
while boolean expression:
    suite
```

while loop

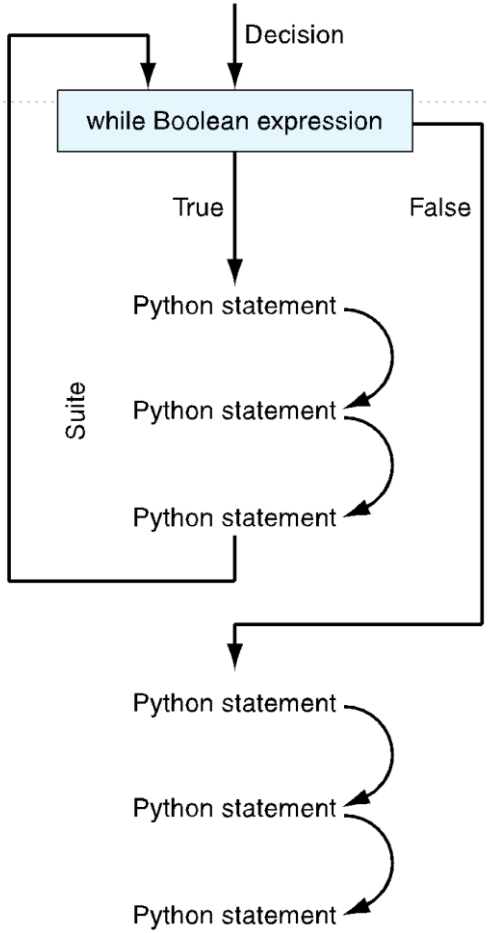


FIGURE 2.4 ***while*** loop.

repeat while the boolean is true

- while loop will repeat the statements in the suite while the boolean is `True` (or its Python equivalent)
- If the Boolean expression never changes during the course of the loop, the loop will continue forever.

Live Coding: Bakso

```
import turtle

t = turtle.Screen()
kura = turtle.Turtle()
jumlah_lingkaran = t.numinput("Contoh Program", "Jumlah lingkaran:")

counter = 0

while counter < int(jumlah_lingkaran):
    kura.circle(20)
    kura.forward(40)
    counter = counter + 1
```

General approach to a `while`

- outside the loop, initialize the boolean
- somewhere inside the loop you perform some operation which changes the state of the program, eventually leading to a False boolean and exiting the loop
- Need to have both!

`for` and iteration

- One of Python's strengths is its rich set of built-in data structures
- The for statement iterates through each element of a collection (list, etc.)

```
for element in collection:
    suite
```

for loop

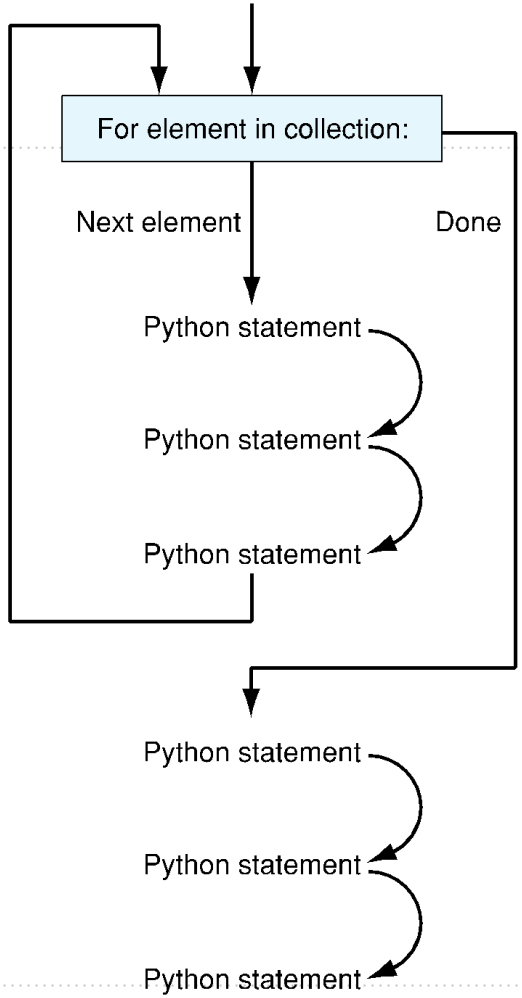


FIGURE 2.5 Operation of a ***for*** loop.

Live Coding: Bakso Besar dan Kecil

```
import turtle
import random

t = turtle.Screen()
kura = turtle.Turtle()
jumlah_lingkaran = t.numinput("Contoh Program", "Jumlah lingkaran:")

counter = 0

while counter < int(jumlah_lingkaran):
    angka_acak = random.randint(1, 100)
    if angka_acak > 50:
        kura.circle(20) # bakso besar
    else:
        kura.circle(10) # bakso kecil
    kura.forward(40)
    counter = counter + 1
```

Developing a `while` loop

Working with the ***loop control variable***:

- **Initialize** the variable, typically outside of the loop and before the loop begins.
- The **condition** statement of the while loop involves a Boolean using the variable.
- **Modify** the value of the control variable during the course of the loop

Issues:

- Loop never starts:

the control variable is not initialized as you thought (or perhaps you don't always want it to start)

- Loop never ends:

the control variable is not modified during the loop (or not modified in a way to make the Boolean come out `False`)

Exercise

Write a program to check whether a certain number given by the user is a **perfect number!**

Hint: A perfect number is a number that is equal to the sum of all of its divisors. For example: 6 = 1+2+3.

`while` loop, round two

- while loop, oddly, can have an associated `else` suite
- `else` suite is executed when the loop finishes **under normal conditions**
  - basically the last thing the loop does as it exits

while with else

```
while booleanExpression:
      suite
      suite
else:
      suite
      suite
rest of the program
```

while-else loop

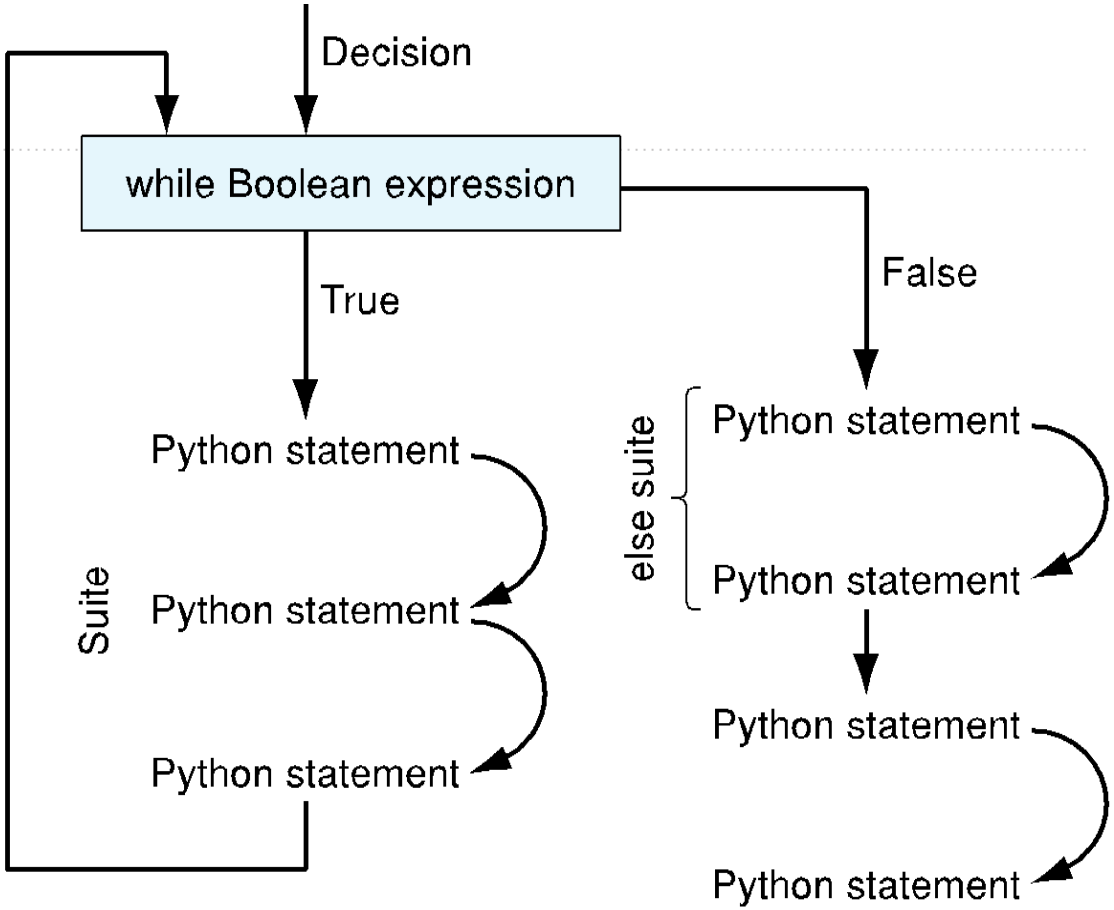


FIGURE 2.9 ***while-else.***

Live Coding: `while-else`

```
counter = 0

while counter < 3:
    print("Inside while with counter:", counter)
    counter = counter + 1
else:
    print("Inside else")
```

Live Coding: Nested `while`

```
counter_i = 0
while counter_i < 3:
    counter_j = 0
    while counter_j < 5:
        print(str(counter_i)+str(counter_j), end = ' ')
        counter_j = counter_j+1
    print()
    counter_i = counter_i+1
```

`break` statement

- A `break` statement in a loop, if executed, exits the loop
- It exits immediately, skipping whatever remains of the loop **as well as the else** statement (if it exists) of the loop
- used for a non-normal exit (early exit) of the loop

Live Coding: `while-else-break`

```
counter = 0

while counter < 3:
    print("Inside while with counter:", counter)
    if counter == 1:
        break
    counter = counter + 1
else:
    print("Inside else")
```

Live Coding: Guess a number

```
import random

number = random.randint(0,10) # get random number between 0-10
guess = int(input("Guess a number (0-10): "))

while 0 <= guess <= 10:
    if guess > number:
        print("Too high")
    elif guess < number:
        print("Too low")
    else:
        print("You guessed it! The number was", number)
        break
    guess = int(input("Guess (again) a number (0-10):"))
else:
    print("Program quit, input is not right!")
```

`continue` statement

- A `continue` statement, if executed in a loop, means to immediately jump back to the top of the loop and re-evaluate the conditional
- Any remaining parts of the loop are skipped for the one iteration when the continue was executed

Live Coding: Masukkan 3 angka genap

```
jumlah_genap = 0

while jumlah_genap < 3:
    angka = int(input("Masukkan angka genap: "))
    if angka % 2 != 0:
        continue
    jumlah_genap += 1
    print("Jumlah sampai saat ini:", jumlah_genap)
```

change in control: `break` and `continue`

- while loops are easiest to read when the conditions of exit are clear
- Excessive use of continue and break within a loop suite make it more difficult to decide when the loop will exit and what parts of the suite will be executed each loop.
- Use them judiciously.

Sentinel loop

- Loop that is controlled by a sentinel (a particular value to terminate the loop).

Example: the following program will keep prompting an input from users until they type “quit”

```
x=input()
while x!=“quit”:
    print(“you entered ”, x)
    x=input()
```

General form:

```
while value != sentinel value:
                # process value
                # get another value
```

While overview

```
while test1:
     # statement_list_1
     if test2:
          break              # Exit loop now; skip else
     if test3:
          continue           # Go to top of loop now
     # more statements
else:
     # statement_list_2      # If we didn't hit a 'break'
```

For overview

- just like the while statement, for statement can also support `else`, `continue`, and `break`.

```
for target in object:
    # statement_list_1
    if test1:
        break              # Exit loop now; skip else
    if test2:
        continue           # Go to top of loop now
    # more statements
else:
    # statement_list_2     # If we didn't hit a 'break'
```

Equivalence of `while` and `for`

- It is possible to write a `while` loop that behaves like a `for` loop.

Exercise: Write an equivalent form of the following code using for loop!

```
for i in range (5):
     print(i)
```

- Not every `while` loop can be expressed as a `for` loop. Example ??
- How to decide which to use, `for` or `while`?

`Range` function

- to generate a **sequence of integers**
- the range function takes 3 arguments:
  - the beginning of the range. Assumed to be **0** if not provided
  - the end of the range, but **not inclusive** (up to but not including the number). Required
  - the step of the range. Assumed to be **1** if not provided
- if only one arg provided, assumed to be the **end value**

Iterating through the sequence

- A range is also called **iterable**, because we can iterate through a range. For example:

```
for num in range(1,5):
    print(num)
```

- range represents the sequence 1, 2, 3, 4
- for loop assigns `num` to each of the values in the sequence, one at a time, in sequence
- prints each number (one number per line)

range generates on demand

Range generates its values on demand

```
>>> range(1,10)
range(1, 10)
>>> my_range=range(1,10)
>>> type(my_range)
<class 'range'>
>>> len(my_range)
9
>>> for i in my_range:
        print(i, end=' ')

1 2 3 4 5 6 7 8 9
>>>
```

Live Coding: Bermain dengan range

```
for i in range(10):
    print(i, end=" ")

print()

for i in range(1,7):
    print(i, end=" ")

print()

for i in range(0,30,5):
    print(i, end=" ")

print()

for i in range(5,-5,-1):
    print(i, end=" ")
```

Live Coding: Bakso versi `for`

```
import turtle

t = turtle.Screen()
kura = turtle.Turtle()
kura.color("gray")
jumlah_lingkaran = t.numinput("Contoh Program", "Jumlah lingkaran:")

for i in range(int(jumlah_lingkaran)):
    kura.begin_fill()
    kura.circle(20)
    kura.end_fill()
    kura.forward(40)
```

Live Coding: Bakso di Mangkok

```
import turtle

t = turtle.Screen()
kura = turtle.Turtle()
kura.color("gray")
jumlah_lingkaran = t.numinput("Bakso di Mangkok", "Jumlah pentol:")

# gambar bakso/pentol
for i in range(0,int(jumlah_lingkaran)):
    heading = kura.heading()
    kura.penup()
    kura.forward(40)
    kura.pendown()
    kura.begin_fill()
    kura.circle(20)
    kura.end_fill()
    kura.penup()
    kura.left(180)
    kura.forward(40)
    kura.setheading(heading)
    kura.left(360/jumlah_lingkaran)
```

Live Coding: Bakso di Mangkok (lanj.)

```
# gambar mangkok
kura.setheading(270)
kura.forward(100)
kura.setheading(0)
kura.color("red")
kura.pendown()
kura.circle(100)

t.exitonclick() # wait for a user click on the canvas
```

Live Coding: Benny EZ Converter v0.1*

- Selalu meminta input sampai keluar

```
print("Selamat datang di Benny™ EZ Converter")
print("Format input [base] [angka]")
print("ketik \"keluar\" untuk keluar program")

while(True): # infinite loop, kecuali ada break
    masukkan = input(">>> ")

    if(masukkan == "keluar"): # jika inputnya "keluar", maka keluar
        print("\n================\nProgram Berhenti")
        break
```

Live Coding: Benny EZ Converter v0.2

- Mencari basis yang tepat

```
...

    masukkan_split = masukkan.split(" ") #split antara [base] dan [angka]

    baseInput = masukkan_split[0] #dapatkan [base]

    if(baseInput == "binary"):
        base = 2
    else:
        base = 8

    print("Basis:", base)
```

Live Coding: Benny EZ Converter v0.2

- Konversi ke desimal sesuai basis

```
...

    # angka yang akan di convert
    numberInput = masukkan_split[1]
    pangkat = len(numberInput) - 1 # pangkat untuk digit paling kiri
    result = 0
    for item in numberInput: # iterasi per karakter di numberInput

        # mencari nilai yang sesuai
        nilai = int(item)
        result += nilai * (base ** pangkat) # nilai dikali base pangkat sekian, masukkan
ke hasil
        pangkat -= 1 # pangkat dikurang 1 tiap iterasi

    print("Dalam desimal:",result)
```

Live Coding: Pertanyaan ganda

```
jawaban = input("Siapa suami dari Raisa?\nA. Jokowi\nB. Keenan\nC. Hamish\n")

# tulis kode di sini
```

Live Coding: Pertanyaan ganda

```
jawaban = input("Siapa suami dari Raisa?\nA. Jokowi\nB. Keenan\nC. Hamish\n")

if jawaban == "C":
    print("Jawaban benar!")
else:
    print("Jawaban salah :(")
```

Live Coding: Pertanyaan ganda

```
jawaban = input("Siapa suami dari Raisa?\nA. Jokowi\nB. Keenan\nC. Hamish\n")

while jawaban != "C":
  print("Jawaban salah :(")
  jawaban = input("Koreksi jawaban: ")
print("Jawaban benar!")
```

Live Coding: Mencari maksimum dari 3 angka input

Live Coding: Mencari maksimum dari 3 angka input

```
x = int(input("x: "))
y = int(input("y: "))
z = int(input("z: "))

if x >= y and x >= z:
  print("Max:", x)
elif y >= x and y >= z:
  print("Max:", y)
else:
  print("Max:", z)
```

Live Coding: Mencari maksimum dari 3 angka input Versi lebih ringkas

```
x = int(input("x: "))
y = int(input("y: "))
z = int(input("z: "))

if x >= y and x >= z:
  print("Max:", x)
elif y >= z:
  print("Max:", y)
else:
  print("Max:", z)
```

Live Coding: Mencari maksimum dari 3 angka input Versi lebih ringkas

```
x = int(input("x: "))
y = int(input("y: "))
z = int(input("z: "))

if y <= x >= z:
  print("Max:", x)
elif y >= z:
  print("Max:", y)
else:
  print("Max:", z)
```

Live Coding: Mencari angka max di list*

```
a_list = [1,5,1,2,4]

max = -1
for elemen in a_list:
  if elemen > max:
    max = elemen

print(max)
```

*** Asumsi elemen di list selalu >= 0, dan list tidak boleh kosong**

Live Coding: Mencari angka min di list*

```
a_list = [1,5,1,2,4]

min = 11
for elemen in a_list:
   # masukkan kode di sini

print(min)
```

*** Asumsi elemen di list selalu >= 0 dan <= 10, dan list tidak boleh kosong**

Live Coding: Mencari angka min di list*

```
a_list = [1,5,1,2,4]

min = 11
for elemen in a_list:
  if elemen < min:
    min = elemen

print(min)
```

*** Asumsi elemen di list selalu >= 0 dan <= 10, dan list tidak boleh kosong**

Live Coding: Berapa angka yang sama dari 3 angka

```
a = int(input())
b = int(input())
c = int(input())

# masukkan kode di sini
```

The program must print one of the numbers: 3 (if all are same), 2 (if two of them are equal to each other and the third is different) or 0 (if all numbers are different).

Live Coding: Berapa yang sama dari 3 angka

```
a = int(input())
b = int(input())
c = int(input())

if a == b == c:
    print(3)
elif a == b or a == c or b == c :
    print(2)
else:
    print(0)
```

The program must print one of the numbers: 3 (if all are same), 2 (if two of them are equal to each other and the third is different) or 0 (if all numbers are different).

Live Coding:
Diberikan angka N, hitung 1 + 2 + ... + N

Live Coding:
Diberikan angka N, hitung 1 + 2 + ... + N (Oops!)

```
n = int(input())

total = 0
for i in range(n):
  total = total + i

print(total)
```

Live Coding:
Diberikan angka N, hitung 1 + 2 + ... + N

```
n = int(input())

total = 0
for i in range(n+1):
  total = total + i

print(total)
```

Live Coding:
Diberikan angka N, hitung N! (= 1 * 2 * … * N) (Oops!)

```
n = int(input())

total = 1
for i in range(n+1):
  total = total * i

print(total)
```

Live Coding:
Diberikan angka N, hitung N! (= 1 * 2 * ... * N)

```
n = int(input())

total = 1
for i in range(1, n+1):
  total = total * i

print(total)
```

Live Coding: Diberikan angka N, gambar piramida setengah seperti ini:

```
          * # sebanyak 1
         ** # sebanyak 2
          .
          .
          .
 *********** # sebanyak N-1
************ # sebanyak N
```

Contoh apabila N = 5:

```
    *
   **
  ***
 ****
*****
```

Hint: Suatu string dapat dicetak secara berulang menggunakan *
Contoh: "a"*3 = "aaa"

Live Coding: Diberikan angka N, gambar piramida setengah!

```
n = int(input())

for i in range(1,n+1):
  print(" "*(n - i) + "*"*(i))
```

Live Coding: Apa yang dilakukan program ini?

```
n = int(input())

for i in range(n):
  print("*"*(i+1))
```

Plotting Data with Pylab

- **Pylab**: a module provided in Python (in addition to the `turtle` module) to plot data in both two and three dimensions
- To use the module, just write "`import pylab`" in the top of your program

First Plot using Pylab

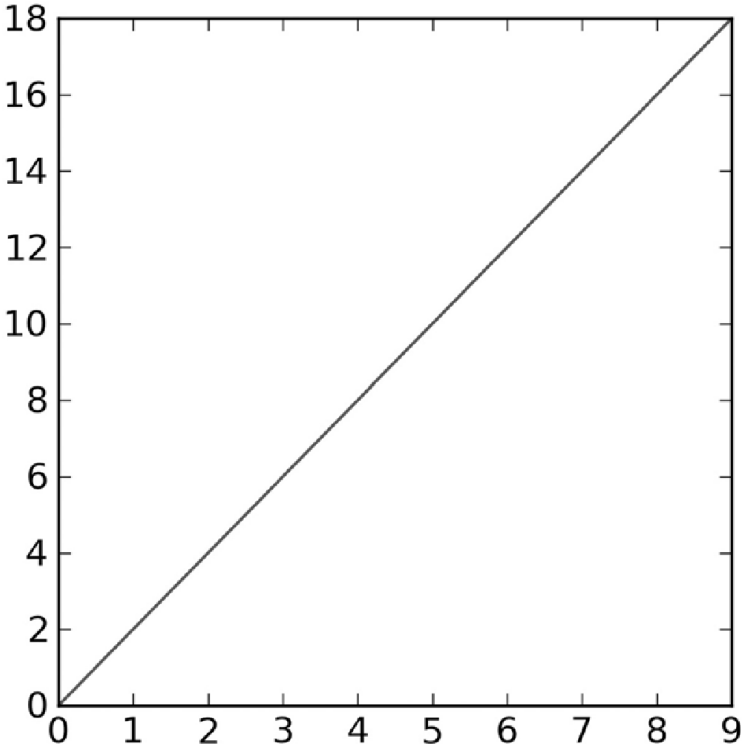
```
import pylab
list_of_ints=[]
for counter in range(10):
     list_of_ints.append(counter*2)

print(list_of_ints)
print(len(list_of_ints))

# now plot the l i s t
pylab.plot(list_of_ints)
pylab.show()
```

This is a `list` data structure (will be explained in more detail later in chapter 7) .
List is just a sequence of objects that are separated by commas. Example: [1, 2, 3, 4]

Function "`append`" will insert the given input to the end of the sequence.

Function "`plot`" uses the value of input sequence (e.g. list_of_ints) as **y-coordinate**. By default, it uses the index of input sequence as **x-coordinate** if it is not given explicitly by users

Output:



More interesting plot: sine wave

```
import math
Import pylab
y_values=[]
x_values=[]
number=0.0

While number < math.pi*4:
     y_values.append(math.sin(number))
     x_values.append(number)
     number+=0.1


# now plot the x and y values as red circles
pylab.plot(x_values, y_values, 'ro')
pylab.show()
```

module “math” is imported to obtain `pi` and `sin(x)` values

- Here we explicitly give the value of list “x_values” as x coordinate, so the **plot** function does not use the default value (Note: compare this with the previous example!)
- “ro” means that the color of the plot is **red**, and the marker of the plot is in the form of **circle**

Output: