

Control Structures

Binary Values

There are 10 kinds of people:

- ▶ those who know binary, and
- ▶ those who don't.

Python Booleans

In Python, boolean values have the `bool` type. Four kinds of boolean expressions:

- ▶ `bool` literals: `True` and `False`
- ▶ `bool` variables
- ▶ expressions formed by combining non-`bool` expressions with comparison operators
- ▶ expressions formed by combining `bool` expressions with logical operators

Comparison Expressions

Simple boolean expressions formed with comparison operators:

- ▶ Equal to: `==`, like `=` in math
 - ▶ Remember, `=` is assignment operator, `==` is comparison operator!
- ▶ Not equal to: `!=`, like `≠` in math
- ▶ Greater than: `>`, like `>` in math
- ▶ Greater than or equal to: `>=`, like `≥` in math

...

Examples:

```
1 == 1 # True
1 != 1 # False
1 >= 1 # True
1 > 1 # False
```

Logical Expressions

Simple boolean expressions can be combined to form larger expressions using logical operators and, or, and not.

```
(1 == 1) and (1 != 1) // False  
(1 == 1) or (1 != 1) // True
```


The if-else Statement

Conditional execution:

```
if boolean_expression:  
    # a single statement executed when boolean_expression is true  
else:  
    # a single statement executed when boolean_expression is false
```

- ▶ `boolean_expression` is not enclosed in parentheses
- ▶ `else:` not required

Example:

```
if (num % 2) == 0:  
    print("I like " + str(num))  
else:  
    print("I'm ambivalent about " + str(num))
```


Blocks

Python is block-structured. Contiguous sequences of statements at the same indentation level form a block. Blocks are like single statements (not expressions - they don't have values).

```
if num % 2 == 0:
    print(str(num) + " is even.")
    print("I like even numbers.")
else:
    print(str(num) + " is odd.");
    print("I'm ambivalent about odd numbers.")
```

Multi-way if-else Statements

This is hard to follow:

```
if color == "red":  
    print("Redrum!")  
else:  
    if color == "yellow":  
        print("Submarine")  
    else:  
        print("A Lack of Color")
```

This multi-way if-else is equivalent, and clearer:

```
if color == "red":  
    print("Redrum!")  
elif color == "yellow":  
    print("Submarine")  
else:  
    print("A Lack of Color")
```

Short-Circuit Evaluation

Here's a common idiom for testing an operand before using it:

```
if (kids != 0) and ((pieces / kids) >= 2):  
    print("Each kid may have two pieces.")
```

In this example Python uses short-circuit evaluation. If

`kids != 0`

evaluates to `False`, then the second sub-expression is not evaluated, thus avoiding a divide-by-zero error.

Loops

Algorithms often call for repeated action, e.g. :

- ▶ “repeat ... while (or until) some condition is true” (looping) or
- ▶ “for each element of this array/list/etc. ...” (iteration)

Python provides two control structures for repeated actions:

- ▶ while loop
- ▶ for iteration statement

while Loops

while loops are pre-test loops: the loop condition is tested before the loop body is executed

```
while condition: # condition is any boolean expression
    # loop body executes as long as condition is true
```

Example

```
>>> def countdown(n):
...     while n > 0:
...         print (n)
...         n -= 1
...
print ('Blast off!')
...
>>> countdown(5)
5
4
3
2
1
Blast off!
```

for Statements

for is an **iteration** statement

- ▶ iteration means visiting the elements of an iterable data structure

In the for loop:

```
>>> animal = 'Peacock'
>>> for animal in ['Giraffe', 'Alligator', 'Liger']:
...     print(animal)
...
Giraffe
Alligator
Liger
>>> animal
'Liger'
```

- ▶ animal is assigned to each element of the iterable list of animals in successive executions of the for loop's body
- ▶ notice that the loop variable re-assigned an existing variable

break and else

- ▶ break terminates execution of a loop
- ▶ optional else clause executes only if loop completes without executing a break (weird - don't ever write code with for~else~)

```
>>> def sweet_animals(animals):  
...     for animal in animals:  
...         print(animal)  
...         if animal == 'Liger':  
...             print('Mad drawing skillz!')  
...             break  
...     else:  
...         print('No animals of note.')  
...  
>>> sweet_animals(['Peacock', 'Liger', 'Alligator'])  
Peacock  
Liger  
Mad drawing skillz!  
>>> sweet_animals(['Peacock', 'Tiger', 'Alligator'])  
Peacock  
Tiger  
Alligator  
No animals of note.
```

Run-time Errors

An error detected during execution is called an exception and is represented at runtime by an exception object. The Python interpreter raises an exception at the point an error occurs. The exception is handled by some exception-handling code. Here we don't handle the `ValueError` ourselves, so it's handled by the Python shell:

```
>>> int('e')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'e'
```

We can handle an exception by enclosing potentially error-raising code in a try block and handling errors in an except clause.

```
try:
    code_that_may_raise_error()
except ExceptionType:
    code_that_handles_exception()
```

`ExceptionType` is optional. If left off, except clause will catch any exception.

Exception Handling Example

```
>>> def get_number_from_user():
...     input_is_invalid = True
...     while input_is_invalid:
...         num = input('Please enter a whole number: ')
...         try:
...             num = int(num)
...             # Won't get here if exception is raised.
...             input_is_invalid = False
...         except ValueError:
...             print(num + ' is not a whole number. Try again.')
...     return num
...
>>> get_number_from_user()
Please enter a whole number: e
e is not a whole number. Try again.
Please enter a whole number: 3
3
```

For more information, see

<https://docs.python.org/3/tutorial/errors.html>