UNIVERSITY *of* TASMANIA

KIT100 PROGRAMMING PREPARATION

Lecture Four:

Managing data and making decisions

Lecture objectives

- Working with Operators
- Performing Calculations
- Decision Making

Using operators

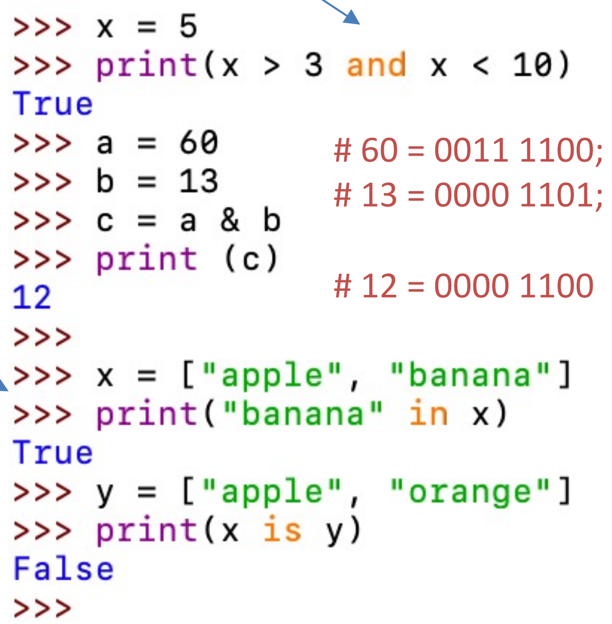
- When we use an **operator**, we must supply either a **variable** or an **expression**.
- Remember that a variable refers to a location (address) in the computer's memory that holds data.
- An **expression** is a combination of variables, operators, constants and functions that result in the creation of a value
  - for example, *1 < 2* is an expression that results in the value True.

Working with operators

- We use operators to define how one piece of data is compared to another, and to modify the information within a variable.
- Operators are essential
  - to perform any sort of math-related task (e.g., **+, - , *, /**)
  - when we are assigning data to variables (e.g., **=**)
  - when comparing data (e.g., **< , >**)
  - making decisions in code (e.g., **==**)

Defining operators

- Python uses the following **categories** of operators:
  - Unary: +, -
  - Arithmetic: *, /
  - Relational: <, >
  - Logical (combine conditional statements): and, or, not
  - Bitwise (compare (binary) numbers): &, |, ~
  - Assignment: =
  - Membership: in, not in
  - Identity: is, is not



```
>>> x = 5
>>> print(x > 3 and x < 10)
True
>>> a = 60          # 60 = 0011 1100;
>>> b = 13          # 13 = 0000 1101;
>>> c = a & b
>>> print (c)
12                  # 12 = 0000 1100
>>>
>>> x = ["apple", "banana"]
>>> print("banana" in x)
True
>>> y = ["apple", "orange"]
>>> print(x is y)
False
>>>
```

Performing calculations

- Math expression: performs a calculation which results in a value that can then be directly used or stored
  - **Arithmetic operator**: performing calculation using numbers
  - **Operands**: the values surrounding operator
    - Variables can be used as operands e.g. `2 + 3.7`
    - Literals can be used as operands e.g. `temperature – heat`
    - Functions (that *return* a value) can be used as operands e.g. `average(1,2,3) / deviation(x)`
    - Expressions can be operands e.g (3 + 1) > (2 * 7) (python evaluates the expressions first before performing the calculation defined by the operator)
  - The resulting value from an expression is typically assigned to a variable, or printed in the output

Arithmetic operators

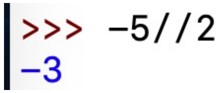
Op	Description	Example	Result
+	Addition - Adds two values together	5+2	7
-	Subtraction - Subtracts the right operand from the left operand	5-2	3
*	Multiplication - Multiplies the left operand by the right operand	5*2	10
/	Float division. Divides the left operand by the right operand	5/2	2.5
%	Modulus - Divides the left operand by the right operand and returns the **remainder**	5%2	1
**	Exponentiation - Calculates the exponential value of the left operand by the right operand	5**2	25
//	Integer division. Divides the left operand by the right operand and only returns a **whole number** value.	5//2	2

Division operators

Notes:

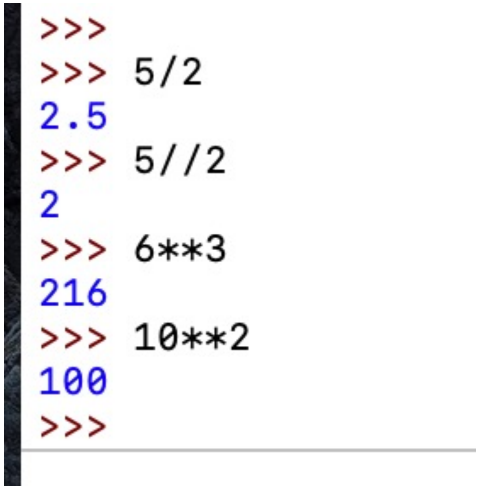
- Two types of **division**:
  1. `/` operator performs **floating point** division
  2. `//` operator performs **integer** division

     Positive results are truncated, negative results are rounded away from zero



```
>>> -5//2
-3
```

- **Exponent operator (`**`)**: Raises a number to a power
  - `a ** b` is equivalent to a^b



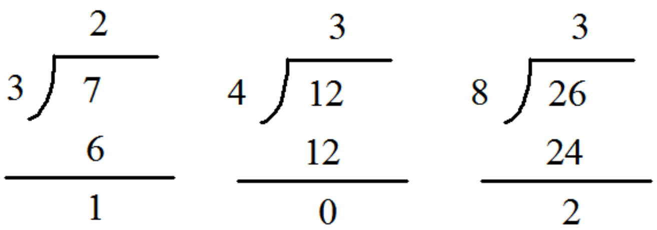
```
>>>
>>> 5/2
2.5
>>> 5//2
2
>>> 6**3
216
>>> 10**2
100
>>>
```

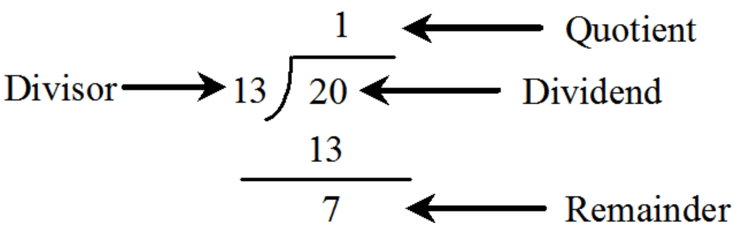
The modulus (remainder) operator

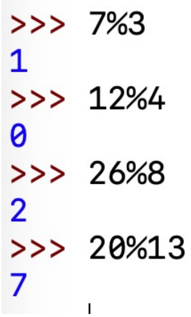
- Modulus operator (`%`): Performs a division and returns the **remainder**
  - e.g., `4%2==0`, `5%2==1`
  - A typical coding use is to detect odd or even numbers

```
result = myNumber%2
```

`result` is 0 if myNumber was an ***even*** *number, 1 if myNumber was* ***odd***



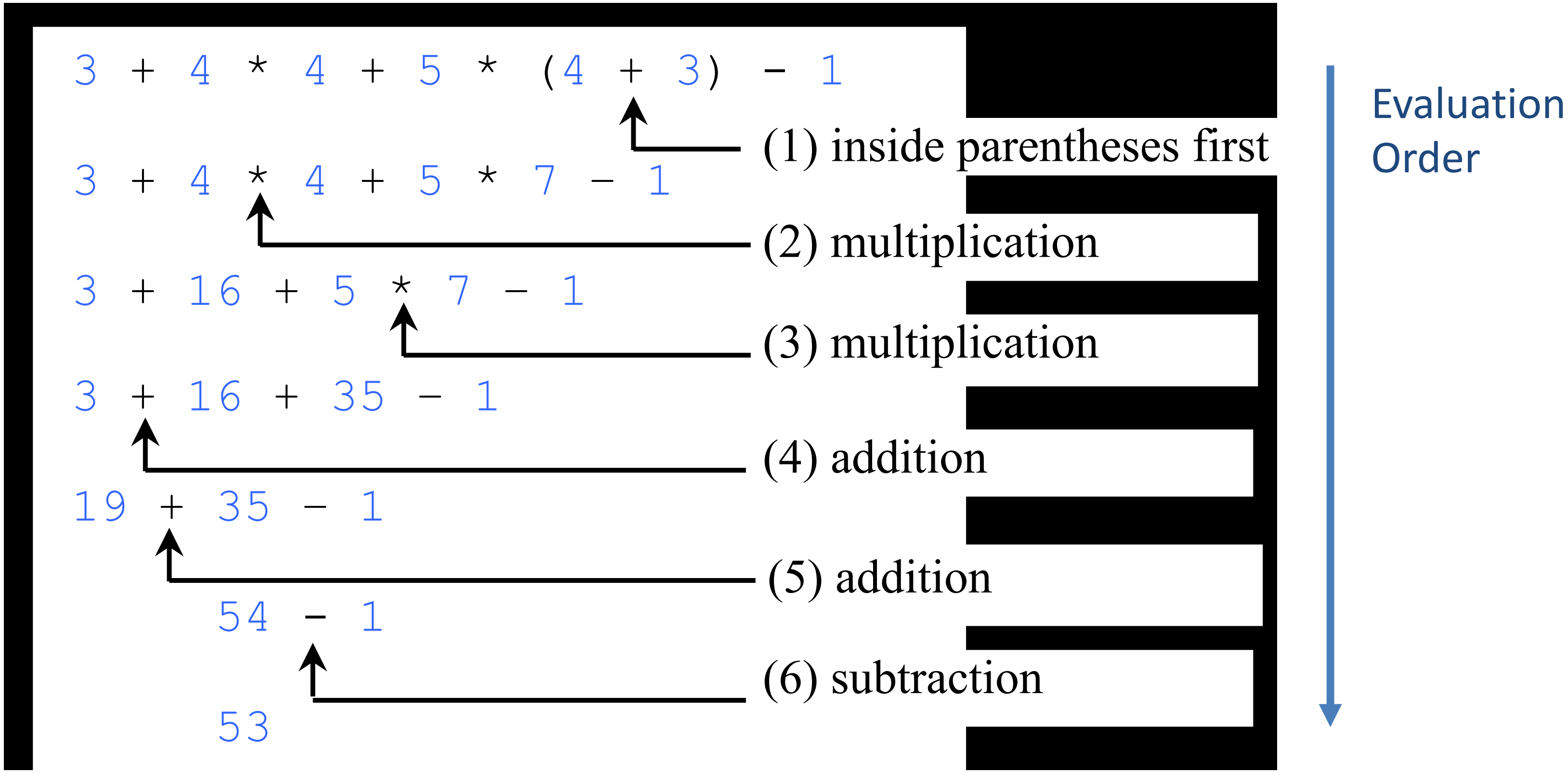




```
>>> 7%3
1
>>> 12%4
0
>>> 26%8
2
>>> 20%13
7
```

How to evaluate an expression

Although Python has its own way to evaluate an expression behind the scenes, the result of a Python expression and its corresponding arithmetic expression are the same. Therefore, you can safely apply the **arithmetic rule** for evaluating a Python expression.



```
3 + 4 * 4 + 5 * (4 + 3) - 1
3 + 4 * 4 + 5 * 7 – 1
3 + 16 + 5 * 7 – 1
3 + 16 + 35 – 1
19 + 35 – 1
54 – 1
53
```

(1) inside parentheses first

(2) multiplication

(3) multiplication

(4) addition

(5) addition

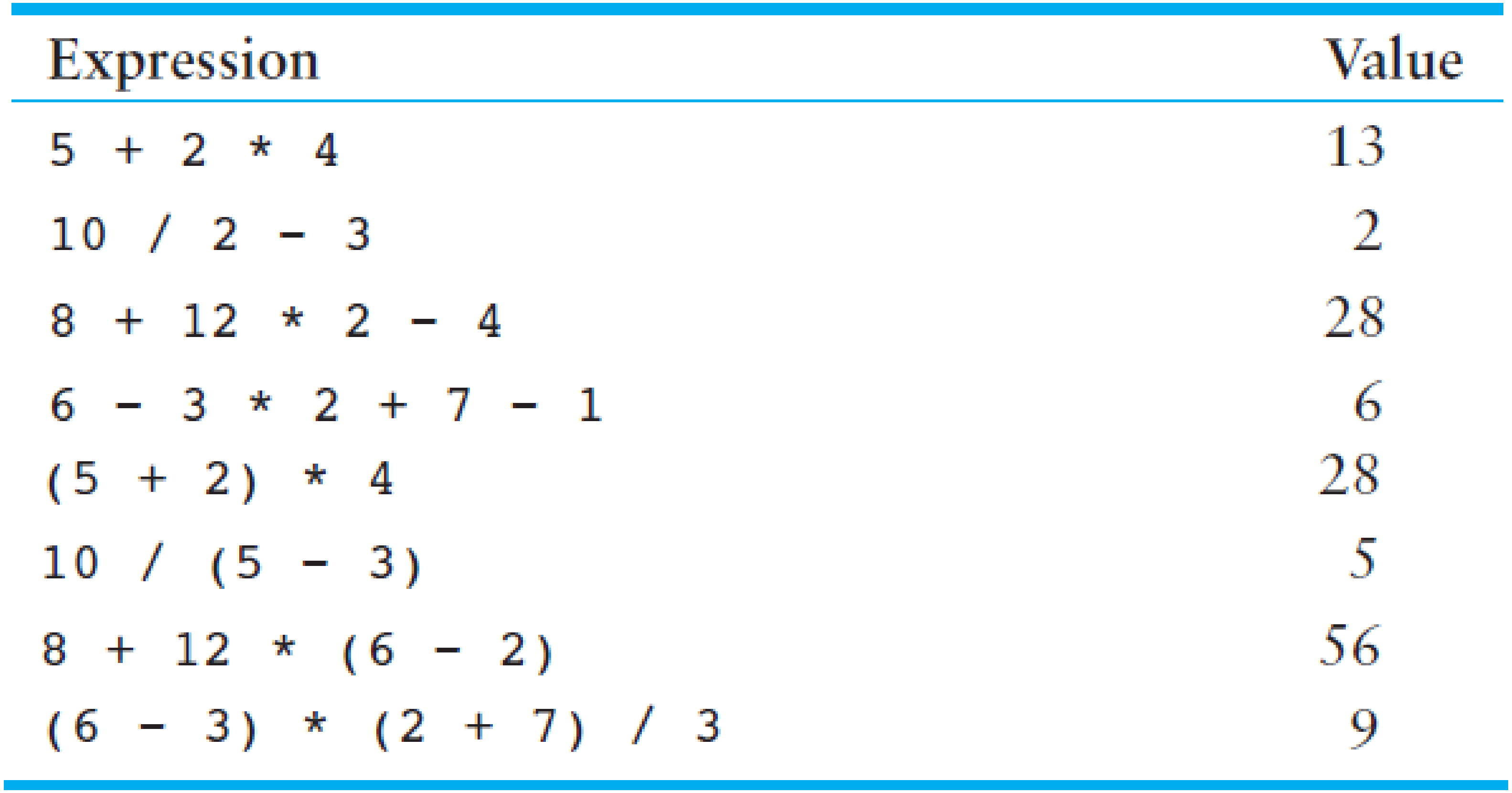
(6) subtraction

Evaluation Order

Operator precedence

- Python operator precedence (the order of operations):
  1. Operations enclosed in parentheses/brackets
     - Forces operations to be performed before others
  2. Exponentiation (**)
  3. Multiplication (*), division (/ and //), and modulus (%)
  4. Addition (+) and subtraction (-)
- Higher precedence operators are performed first
  - The same precedence operators execute **from left to right** **(all have equal precedence)**

Expression examples



Expression	Value
`5 + 2 * 4`	13
`10 / 2 - 3`	2
`8 + 12 * 2 - 4`	28
`6 - 3 * 2 + 7 - 1`	6
`(5 + 2) * 4`	28
`10 / (5 - 3)`	5
`8 + 12 * (6 - 2)`	56
`(6 - 3) * (2 + 7) / 3`	9

Expression examples

$$\frac{3+4x}{5}-\frac{10(y-5)(a+b+c)}{x}+9(\frac{4}{x}+\frac{9+x}{y})$$

is translated to

```
(3+4*x)/5 – 10*(y-5)*(a+b+c)/x + 9*(4/x + (9+x)/y)
```

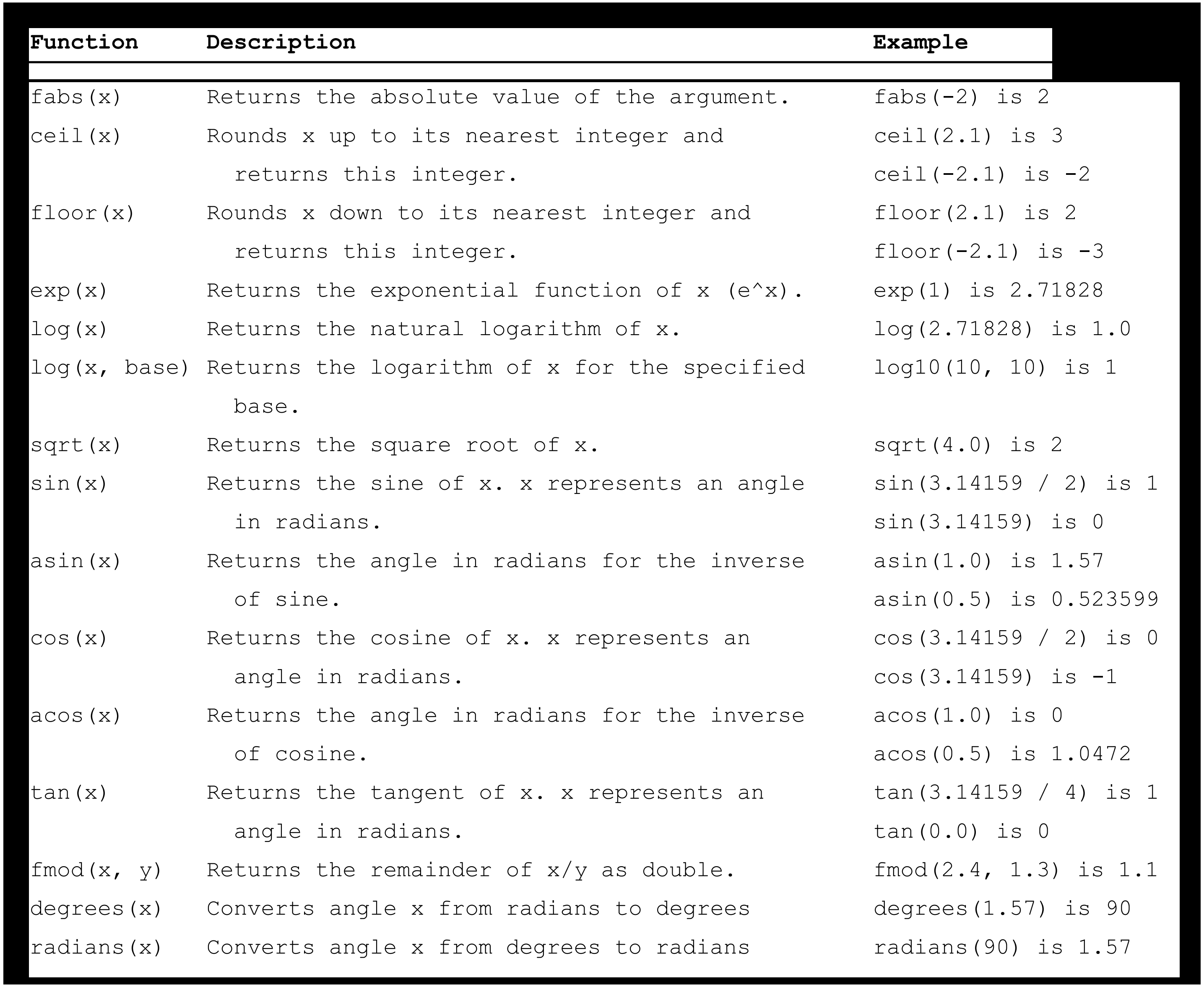
in python

Built-in functions and math module

max, round, abs, pow

```
>>> max(2, 3, 4) # Returns a maximum number
4
>>> min(2, 3, 4) # Returns a minimum number
2
>>> round(3.51) # Rounds to its nearest integer
4
>>> round(3.4) # Rounds to its nearest integer
3
>>> abs(-3) # Returns the absolute value
3
>>> pow(2, 3) # Same as 2 ** 3 (exponentiation function)
8
```

Some math functions



Function	Description	Example
fabs(x)	Returns the absolute value of the argument.	fabs(-2) is 2
ceil(x)	Rounds x up to its nearest integer and returns this integer.	ceil(2.1) is 3 ceil(-2.1) is -2
floor(x)	Rounds x down to its nearest integer and returns this integer.	floor(2.1) is 2 floor(-2.1) is -3
exp(x)	Returns the exponential function of x (e^x).	exp(1) is 2.71828
log(x)	Returns the natural logarithm of x.	log(2.71828) is 1.0
log(x, base)	Returns the logarithm of x for the specified base.	log10(10, 10) is 1
sqrt(x)	Returns the square root of x.	sqrt(4.0) is 2
sin(x)	Returns the sine of x. x represents an angle in radians.	sin(3.14159 / 2) is 1 sin(3.14159) is 0
asin(x)	Returns the angle in radians for the inverse of sine.	asin(1.0) is 1.57 asin(0.5) is 0.523599
cos(x)	Returns the cosine of x. x represents an angle in radians.	cos(3.14159 / 2) is 0 cos(3.14159) is -1
acos(x)	Returns the angle in radians for the inverse of cosine.	acos(1.0) is 0 acos(0.5) is 1.0472
tan(x)	Returns the tangent of x. x represents an angle in radians.	tan(3.14159 / 4) is 1 tan(0.0) is 0
fmod(x, y)	Returns the remainder of x/y as double.	fmod(2.4, 1.3) is 1.1
degrees(x)	Converts angle x from radians to degrees	degrees(1.57) is 90
radians(x)	Converts angle x from degrees to radians	radians(90) is 1.57

Using math functions

- To use a function in the *math* **library** the first statement in your program must be:

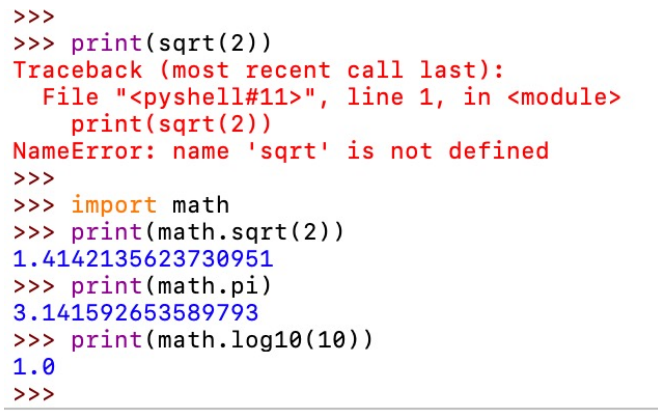
```
import math
```

- The *math* library not only has functions but also useful constants like π and e.
- *To use the functions or the constants* in your program you must apply the **dot** (.) operator. The general syntax for usage is:

`math.function()` **or** `math.constant`

e.g. `math.sqrt(2)` `square root` $\sqrt{2}$

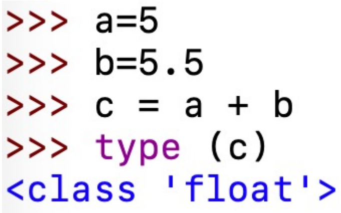
`math.pi`



```
>>>
>>> print(sqrt(2))
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    print(sqrt(2))
NameError: name 'sqrt' is not defined
>>>
>>> import math
>>> print(math.sqrt(2))
1.4142135623730951
>>> print(math.pi)
3.141592653589793
>>> print(math.log10(10))
1.0
>>>
```

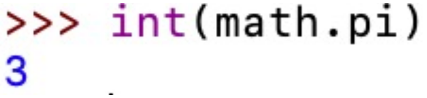
Mixed-type expressions

- The data type resulting from a math operation depends on the data types of the operands
  - Two `int` values: result is an `int`
  - Two `float` values: result is a `float`
  - `int` and `float`: `int` is temporarily converted to `float`, result of the operation is a `float`
    - *This is a "mixed-type"* expression



```
>>> a=5
>>> b=5.5
>>> c = a + b
>>> type (c)
<class 'float'>
```

  - The type conversion of `float` to `int` causes a truncation of the fractional part e.g. 3.1415 → 3



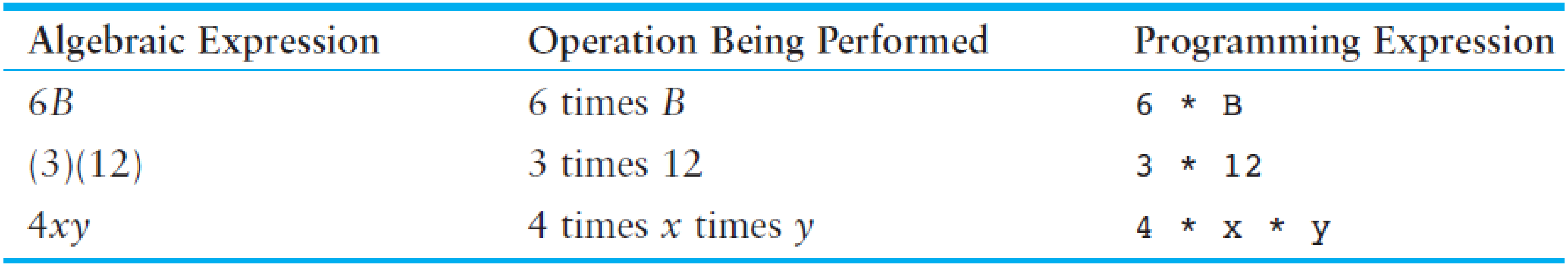
```
>>> int(math.pi)
3
```

Converting math formulas to Python

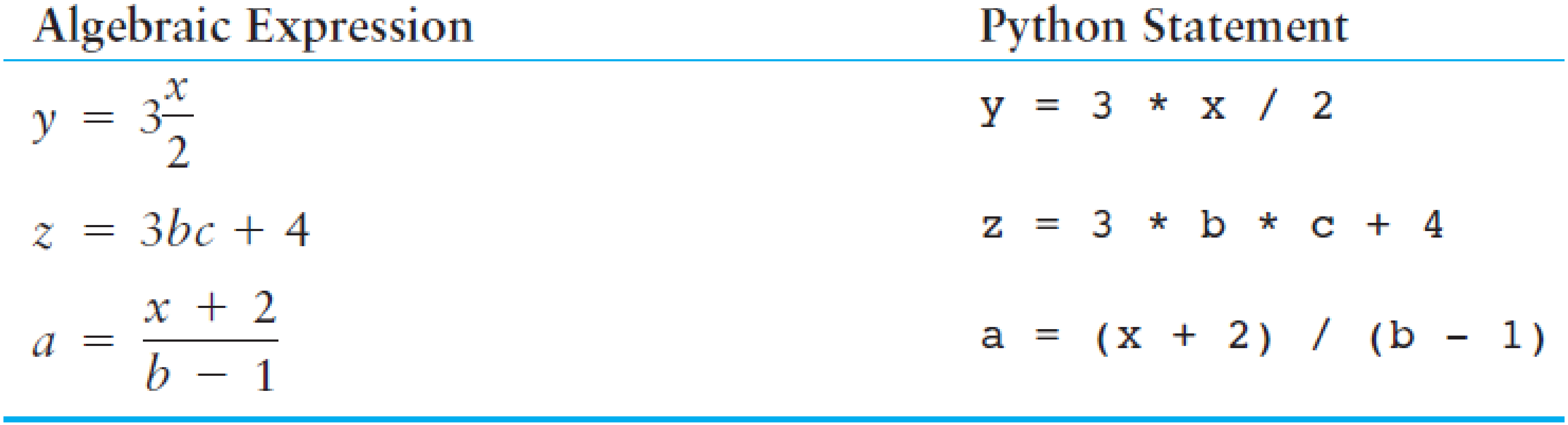
- An operator is required for any mathematical operation
- When converting a mathematical expression to a programming statement:
  - May need to add multiplication operators

    (the multiplication may be implicit in the mathematical expression, e.g. $a(b+c)$ means $a \times (b + c)$)
  - May need to insert parentheses/ brackets (to force operator precedence)

Converting math formulas to Python



Algebraic Expression	Operation Being Performed	Programming Expression
$6B$	6 times B	`6 * B`
$(3)(12)$	3 times 12	`3 * 12`
$4xy$	4 times x times y	`4 * x * y`

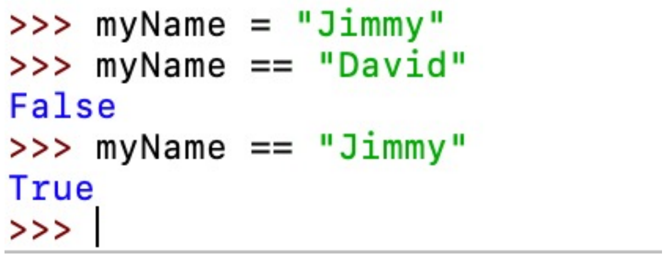


Algebraic Expression	Python Statement
$y = 3\frac{x}{2}$	`y = 3 * x / 2`
$z = 3bc + 4$	`z = 3 * b * c + 4`
$a = \frac{x + 2}{b - 1}$	`a = (x + 2) / (b - 1)`

Relational operators

- A relational operator compares **two** values and the result specifies whether the relationship is true (or false).
- For example: **1** is less than **2**, but **1** is never greater than **2**.
- The ***truth*** value of relations is often used to make decisions in programs to ensure that a condition for a specific task is met.
  - `1>2` is **`False`**
  - `16>4` is **`True`**
  - `myName = “Jimmy”` # (assign ”Jimmy“ to myName)
    `myName == “Jimmy”` is **`True`**

Does it equal?



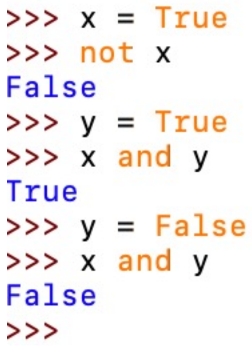
```
>>> myName = "Jimmy"
>>> myName == "David"
False
>>> myName == "Jimmy"
True
>>> |
```

Relational operators

Operator	Description	Example
`==`	Determines if two values are **equal**.	`1==2` is `False`
`!=`	Determines if two values are **not equal**.	`1!=2` is `True`
`>`	Tells you if the left operand is greater than the right operand value.	`1>2` is `False`
`<`	Tells you if the left operand is smaller than the right operand value.	`1<2` is `True`
`>=`	Tells you if the left operand value is of greater **or** equal value to the right operand value.	`1>=2` is `False`
`<=`	Tells you if the left operand value is of lesser **or** equal value to the right operand value.	`1<=2` is `True`

Logical operators

- Logical operators combine True or False values in some way that results in a True or False value.
- We use logical operators to create **Boolean expressions**, usually to help us to **decide** when we want a program to perform certain tasks.
  - **and** operator, **or** operator: these are **binary** operators (2 operands needed), they "connect" two **Boolean** expressions into a ***compound Boolean expression***
  - **not** operator: this is a **unary** operator (one operand needed), it reverses the truth of its **Boolean** operand (True→False, False→True)



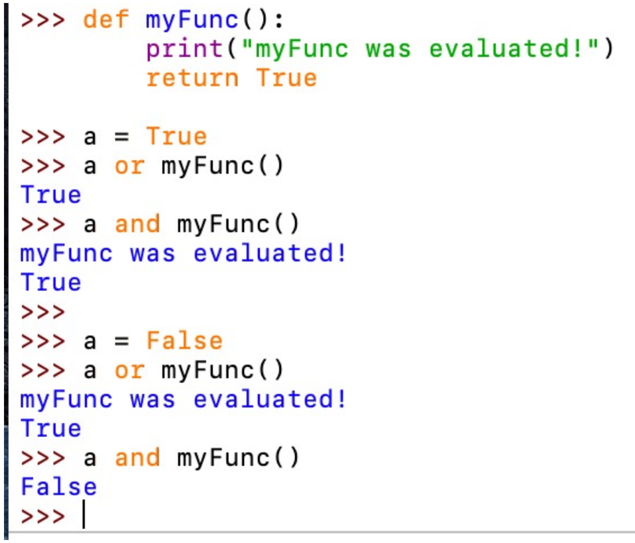
```
>>> x = True
>>> not x
False
>>> y = True
>>> x and y
True
>>> y = False
>>> x and y
False
>>>
```

Logical operators

Operator	Description	Example	Result
`and`	Determines whether both operands are true.	`True and True`	`True`
		`True and False`	`False`
		`False and True`	`False`
		`False and False`	`False`
`or`	Determine when one of two operands is true.	`True or True`	`True`
		`True or False`	`True`
		`False or True`	`True`
		`False or False`	`False`
`not`	Negates the truth value of a single operands. A true value becomes false and a false value becomes true.	`not True`	`False`
		`not False`	`True`

Short-circuit evaluation

- **Short circuit evaluation**: This is when we decide the value of a **compound Boolean** expression after only evaluating **one** sub-expression. Python evaluates from left to right and stops evaluating if the result is "already known"
- Performed by the `or` and `and` operators
  - The `or` operator:
    - ○ If left operand evaluates to **True**, the compound expression must be **True** (*irrespective of the right operand – there's no need to evaluate the right operand so Python doesn't!*).
    - ○ Otherwise, we must evaluate the right operand
  - The `and` operator:
    - ○If left operand evaluates to **False**, the compound expression must be **False** (*irrespective of the right operand – there's no need to evaluate the right operand so Python doesn't!*).
    - ○Otherwise, we must evaluate the right operand



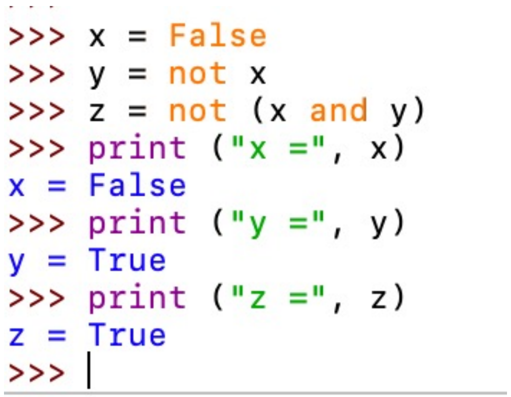
```
>>> def myFunc():
        print("myFunc was evaluated!")
        return True

>>> a = True
>>> a or myFunc()
True
>>> a and myFunc()
myFunc was evaluated!
True
>>>
>>> a = False
>>> a or myFunc()
myFunc was evaluated!
True
>>> a and myFunc()
False
>>> 
```

The not operator

- Takes **one Boolean** expression as an operand and reverses its logical value
  - Sometimes it may be necessary to place parentheses around an expression to clarify which part you are applying the **not** operator to.
  - E.g.

What is output?

```
x = False
y = not x
z = not (x and y)
print("x =",x)
print("y =",y)
print("z =",z)
```



```
>>> x = False
>>> y = not x
>>> z = not (x and y)
>>> print ("x =", x)
x = False
>>> print ("y =", y)
y = True
>>> print ("z =", z)
z = True
>>> |
```

Checking numeric ranges

- To determine whether a numeric value is **within** (between) a specific **range of values**, use the **and** operator:

  Example: `x >= 10 and x <= 20`

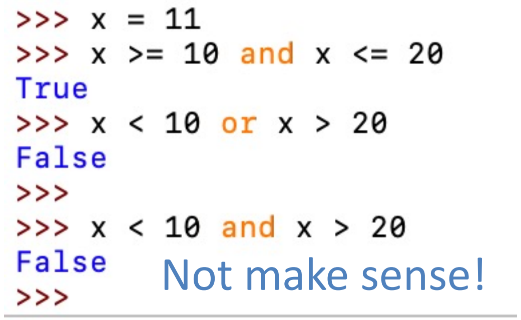
  "x is between 10 and 20 (inclusive)"

- To determine whether a numeric value is **outside** of a specific **range of values**, use the **or** operator:*

  Example: `x < 10 or x > 20`

  "x does not fall between 10 and 20"

**a common mistake is to use and here.. Why?*



```
>>> x = 11
>>> x >= 10 and x <= 20
True
>>> x < 10 or x > 20
False
>>>
>>> x < 10 and x > 20
False
>>>
```

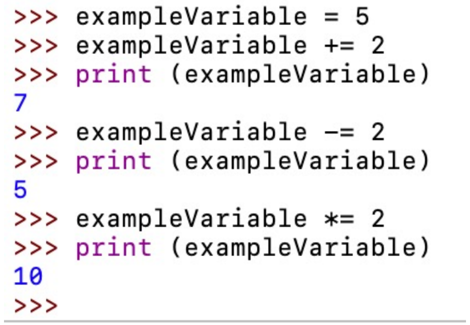
Not make sense!

Assignment operators

- As we have seen in last week's lecture, assignment operators place or store data within a variable.
- We have seen the normal assignment operator `=` already.
  - Please note it is the right-to-left execution rule.
- But there are other assignment operators that can perform mathematical tasks during assignment of data.

```
+=, -=, *=
```

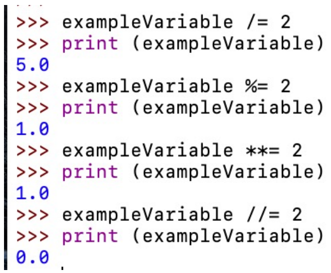
Assignment operators

Operator	Description	Example	Result
=	Assigns the value found in the right operand to the left operand.	`exampleVariable = 5`	`exampleVariable` contains 5
+=	Adds the value found in the right operand to the value found in the left operand and places the result in the left operand.	`exampleVariable += 2`	`exampleVariable` now contains 7
−=	Subtracts the value found in the right operand from the value found in the left operand and places the result in the left operand.	`exampleVariable −= 2`	`exampleVariable` now contains 5
*=	Multiplies the value found in the right operand by the value found in the left operand and places the result in the left operand.	`exampleVariable *= 2` 	`exampleVariable` now contains 10

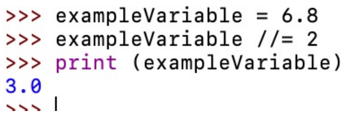
```
>>> exampleVariable = 5
>>> exampleVariable += 2
>>> print (exampleVariable)
7
>>> exampleVariable -= 2
>>> print (exampleVariable)
5
>>> exampleVariable *= 2
>>> print (exampleVariable)
10
>>>
```

Assignment operators continued…

Operator	Description	Example	Result
/=	Divides the value found in the left operand by the value found in the right operand and places the **result** in the left operand.	`exampleVariable /= 2`	`exampleVariable` now contains 5.0 (float type)
%=	Divides the value found in the left operand by the value found in the right operand and places the **remainder** in the left operand.	`exampleVariable %= 2`	`exampleVariable` now contains 1.0
**=	Determines the **exponential value** found in the left operand when raised to the power of the value found in the right operand and places the result in the left operand.	`exampleVariable **= 2`	`exampleVariable` now contains 1.0
//=	Divides the value found in the left operand by the value found in the right operand and places the **integer (whole number)** result in the left operand.	`exampleVariable //= 2`	`exampleVariable` now contains 0.0



```
>>> exampleVariable /= 2
>>> print (exampleVariable)
5.0
>>> exampleVariable %= 2
>>> print (exampleVariable)
1.0
>>> exampleVariable **= 2
>>> print (exampleVariable)
1.0
>>> exampleVariable //= 2
>>> print (exampleVariable)
0.0
```



```
>>> exampleVariable = 6.8
>>> exampleVariable //= 2
>>> print (exampleVariable)
3.0
>>>
```

Assignment operators continued…

Operator	*Example*	*Equivalent*
`+=`	`i += 8`	`i = i + 8`
`-=`	`f -= 8.0`	`f = f - 8.0`
`*=`	`i *= 8`	`i = i * 8`
`/=`	`i /= 8`	`i = i / 8`
`%=`	`i %= 8`	`i = i % 8`

Membership operators

- The membership operators (`in, not in`) detect the appearance of a value within a **list** or sequence and then output the truth value (`True` or `False`) of that appearance.

- *For example, think of the membership operators as you would say comparing your name to a list of names who are granted access to an exclusive club.*
  - *If your name's on the list (`True`), you're allowed in. If it isn't on the list (`False`), you're not allowed in.*

Membership operators

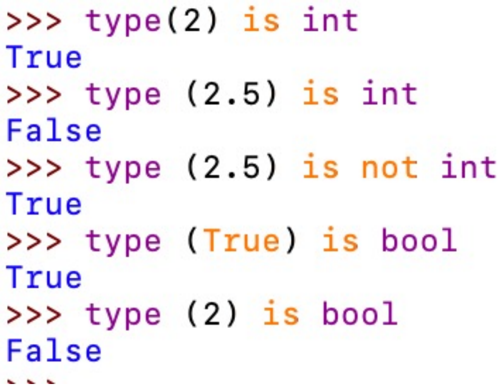
Operator	Description	Example	Result
`in`	Determines whether the value in the left operand appears in the sequence found in the right operand.	`"Hello" in "Hello Goodbye"`	`True`
`not in`	Determines whether the value in the left operand is missing from the sequence found in the right operand.	`"Hello" not in "Hello Goodbye"`	`False`

Example	Result
`"a" in "abcdef"`	`True`

Identity operators

- Identity operators (`is, is not`) determine if a value or an expression is of a certain class, or type.
- We use the identity operators to ensure we are actually working with the type of data we think we are.
- It helps us to avoid errors in our applications and determine the type of processes a value will require.
  - For example, if we are expecting an integer value as an operand, but are given a string instead, we can make our program more robust by checking we have an integer ***before*** we try to use it.

Operator	Description	Example	Result
`is`	Calculates to true when the type of value or expression in the right operand matches the type specified in the left operand.	`type(2) is int`	`True`
`is not`	Calculates to true when the type of value or expression in the right operand does not match the type specified in the left operand.	`type(2) is not int`	`False`

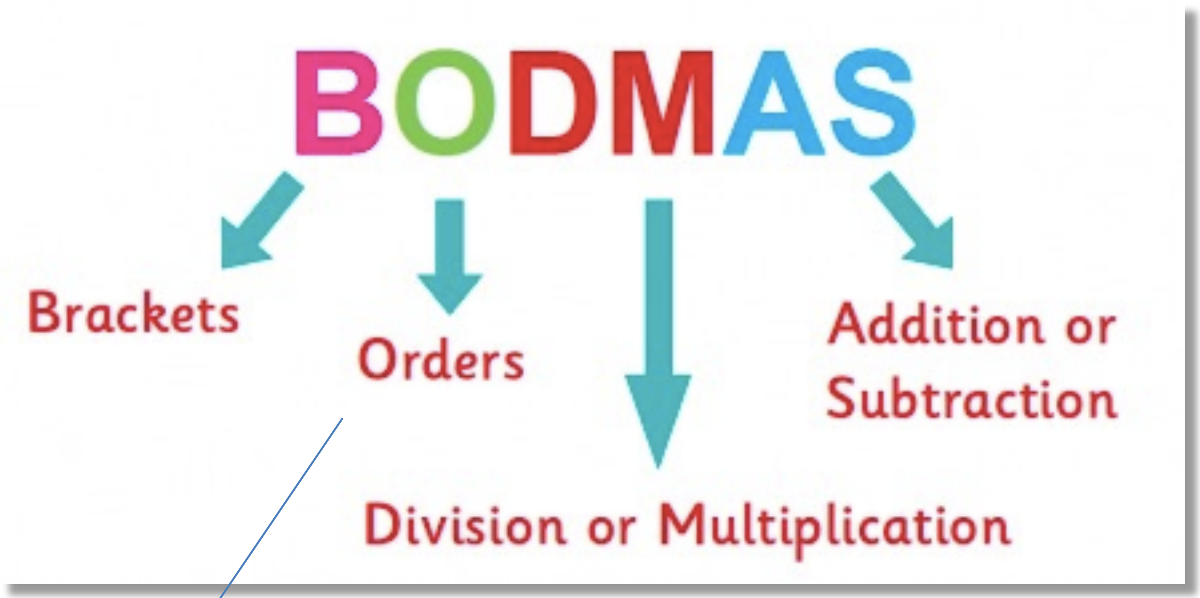


```
>>> type(2) is int
True
>>> type (2.5) is int
False
>>> type (2.5) is not int
True
>>> type (True) is bool
True
>>> type (2) is bool
False
...
```

Python operator precedence

Precedence

Bracket first



Orders of Power (squared etc) – also called exponentiation

Python operator precedence

1. ( )
2. **
3. not
4. *, /, //, %
5. +, -
6. <, <=, >, >=
7. ==, !=
8. and
9. or
10. =, +=, -=, *=, /=, //=, %=

Operator precedence and associativity

- The expression in the parentheses/brackets is evaluated first.
  - (Parentheses can be nested, in which case the expression in the inner parentheses is executed first.)
  - When evaluating an expression without parentheses, the operators are applied according to the precedence rule and the **associativity rule**
    - *left-associative rule excepting assignment operators (see the next slide).*

- If operators with the same precedence are next to each other, their associativity determines the order of evaluation.

Operator associativity

All **equal-precedence** (e.g. + and -, * and /) binary operators (except assignment operators) are ***left-associative***.

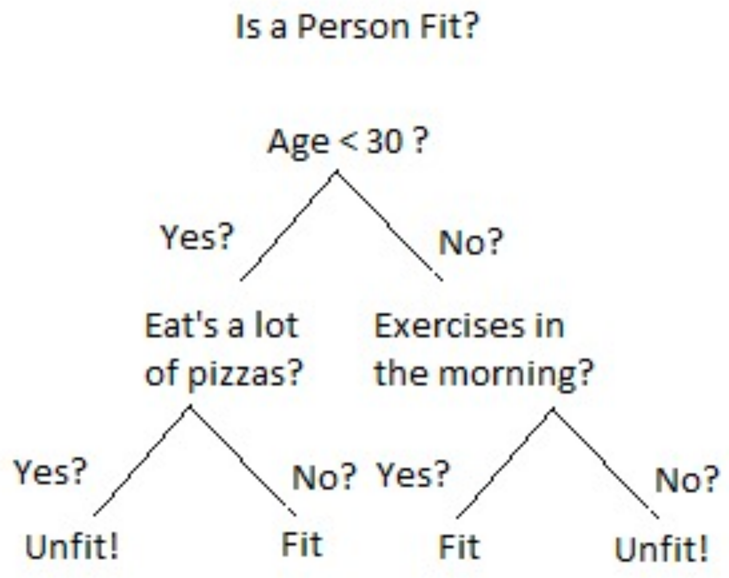
`a - b + c - d` is equivalent to `((a - b) + c) - d`

Assignment operators are ***right-associative***. Therefore, the expression

`a = b += c = 5` is equivalent to `a = (b += (c = 5))`

Making decisions

- The ability to make a decision, to take one path or another, is an essential element of performing useful work.
- Any programming language you may use will include the capability to make decisions in some manner.
  - This is known as *branching*.



Is a Person Fit?

Age < 30 ?

Yes?

No?

Eat's a lot of pizzas?

Exercises in the morning?

Yes?

No?

Yes?

No?

Unfit!

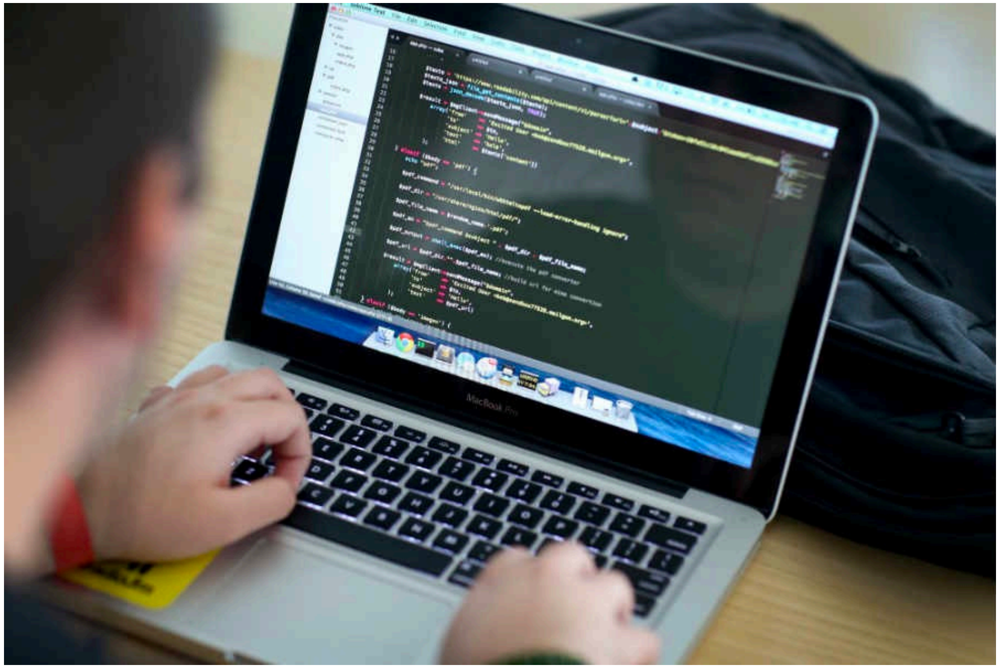
Fit

Fit

Unfit!

Decision making with computers

1. Obtain the actual or current value of something.
2. Compare the actual or current value to a desired value.
3. Perform an action that corresponds to the desired outcome of the comparison.



Making simple decisions using ‘if’

- The ***`if`*** statement is the easiest method for making a decision in Python.
- The ***`if`*** statement simply states that if something is true, Python should perform the *indented* steps that follow.
- We use ***`if`*** statements regularly in everyday life.

Example: *If it’s Monday at 1pm, I’ll attend the KIT001 Lecture.*

The if statement

Prompt

- Python syntax:

```
if condition:
    statement
    statement
```

indent

indent

*Indented statements (1 or more) form a **block** of statements*

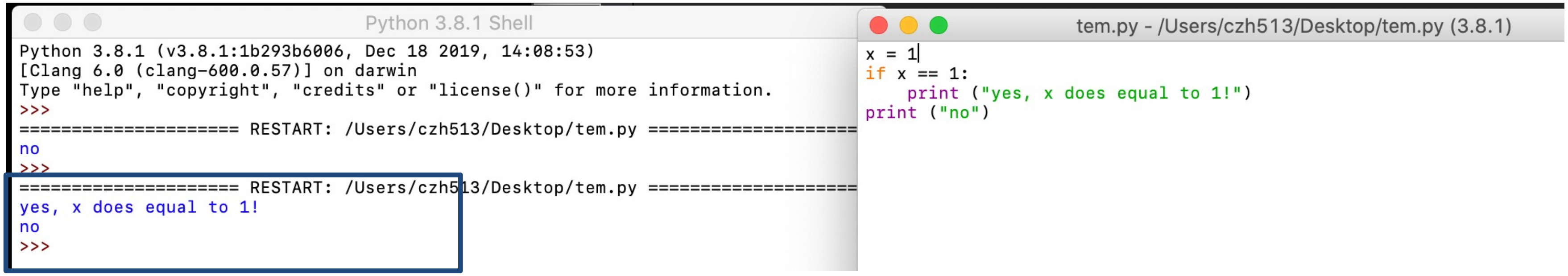
- The first line is known as the `if` clause
  - It includes the keyword `if` followed by a *condition* and a colon/prompt (`:`)
    - The **condition** can evaluate to `True` or `False`
    - When the `if` statement executes, the condition is tested, and if it is **true** the block of indented statements are **executed**. Otherwise, the block of statements are skipped (examples attached)

The if statement

Examples:

When the **if** statement executes, the condition is tested,

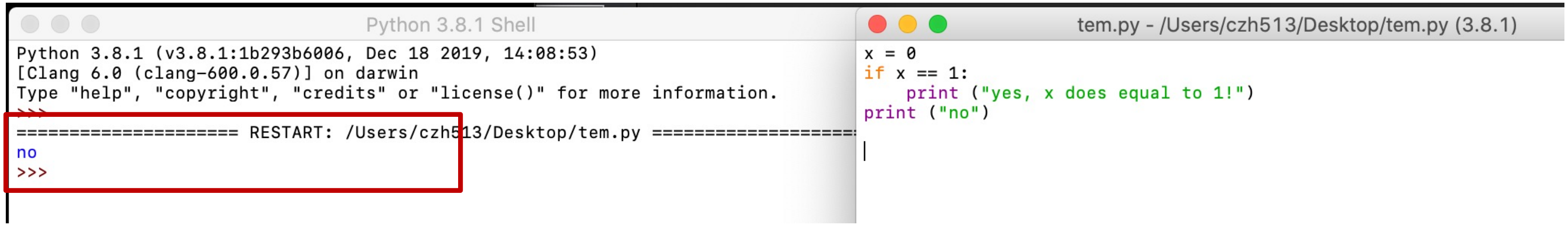
If it is true, the block of indented statements are executed.



```
Python 3.8.1 (v3.8.1:1b293b6006, Dec 18 2019, 14:08:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
==================== RESTART: /Users/czh513/Desktop/tem.py ====================
no
>>>
==================== RESTART: /Users/czh513/Desktop/tem.py ====================
yes, x does equal to 1!
no
>>>
```

```
x = 1
if x == 1:
    print ("yes, x does equal to 1!")
print ("no")
```

Otherwise, the block of statements are skipped.



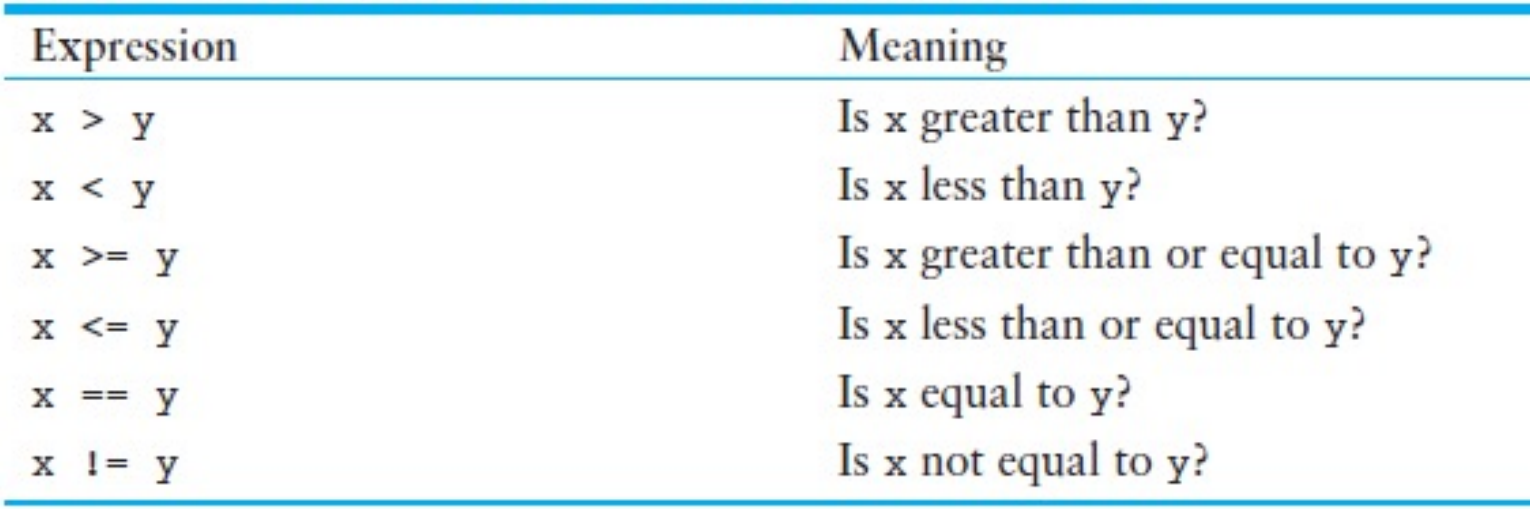
```
Python 3.8.1 (v3.8.1:1b293b6006, Dec 18 2019, 14:08:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
==================== RESTART: /Users/czh513/Desktop/tem.py ====================
no
>>>
```

```
x = 0
if x == 1:
    print ("yes, x does equal to 1!")
print ("no")
```

Boolean expressions and relational operators

- **Boolean expression**: an expression tested by an `if` statement to determine if it is `True` or `False`
  - Example: `a > b`
    - `True` if a is greater than b; `False` otherwise
- **Relational operator**: determines the truth of whether a specific relationship exists between **two** values
  - Example: *greater than* (`>`)

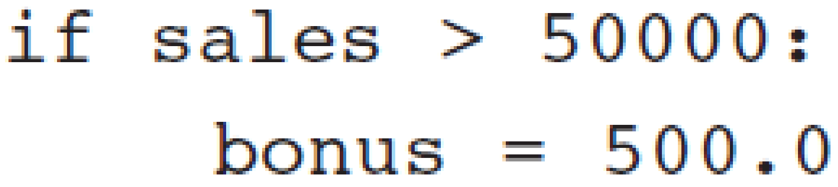
Boolean expressions and relational operators



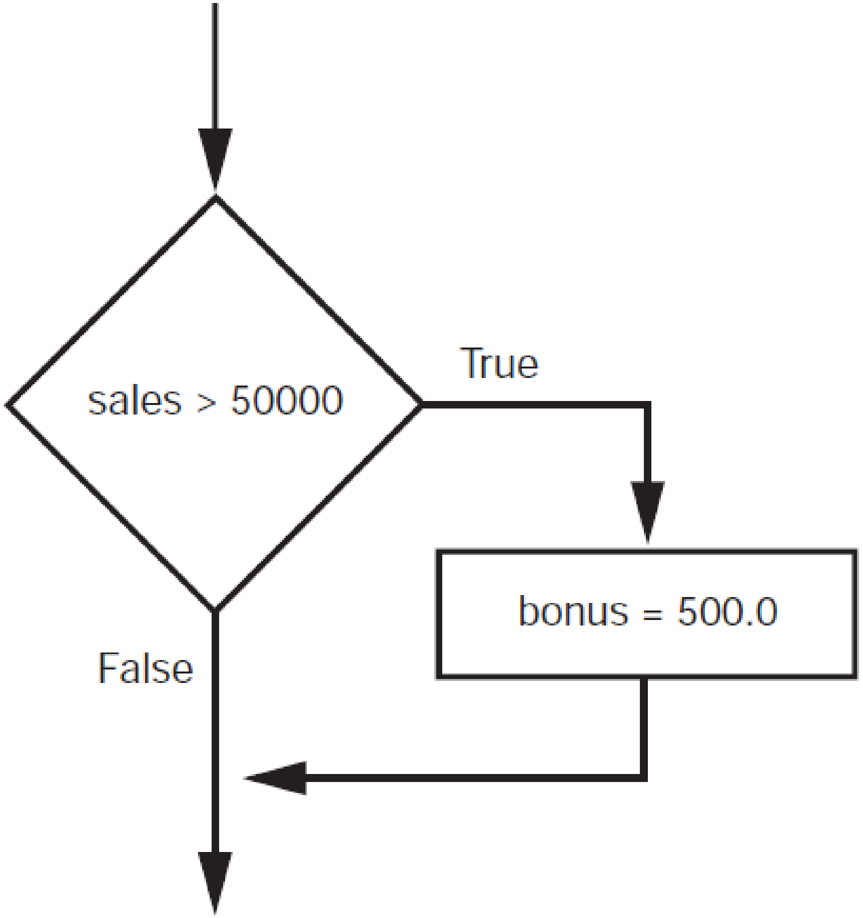
Expression	Meaning
`x > y`	Is x greater than y?
`x < y`	Is x less than y?
`x >= y`	Is x greater than or equal to y?
`x <= y`	Is x less than or equal to y?
`x == y`	Is x equal to y?
`x != y`	Is x not equal to y?

if statement example

When Python sees *if*, it knows that you want it to make a decision.



```
if sales > 50000:
    bonus = 500.0
```



A flowchart may be helpful?

if statement example

```
if sales == 50000:
    bonus = 500.0
```

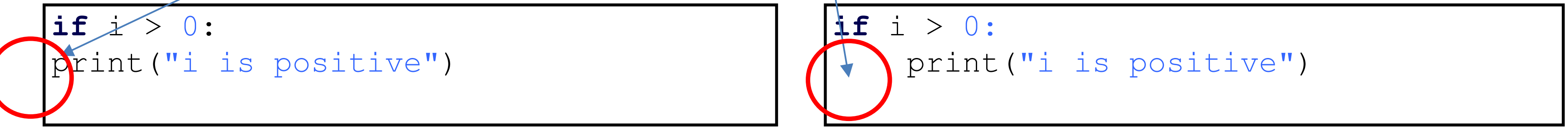
means you want Python to determine whether `sales` is equal to `50000`. Note we used `==` here instead of `=` (which would be a semantic error!)

Using the if statement in an application

- It’s possible to use the ***if*** statement in a number of ways in Python.
- Three common ways:
  - Use a **single** condition to execute a **single** statement when the condition is true.
  - Use a **single** condition to execute **multiple** statements when the condition is true.
  - **Combine** **multiple** conditions into a **single** decision and execute **one or more** statements when the combined condition is true.

Performing multiple tasks

- Sometimes we will want to perform **more than one** task after making a decision.
- Python **relies on indentation** to determine when to **stop** executing tasks as part of an ***if*** statement. When the next line is **out dented** (e.g., >>> in the IDLE), it becomes the first line of code **outside** the *if* block.



```
if i > 0:
print("i is positive")
```

(a) Wrong

```
if i > 0:
    print("i is positive")
```

(b) Correct

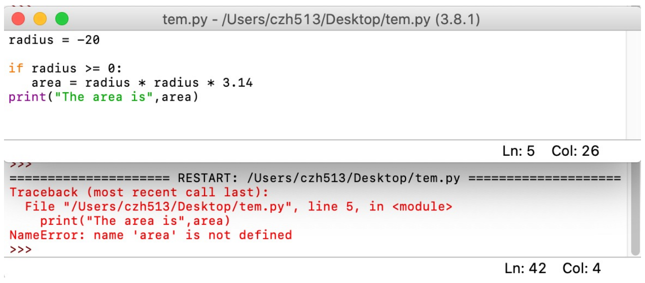
Common errors

The most common errors in selection statements are caused by incorrect indentation. Consider the following code in (a) and (b).

(a)

```
radius = -20

if radius >= 0:
    area = radius * radius * 3.14
print("The area is",area)
```

✗



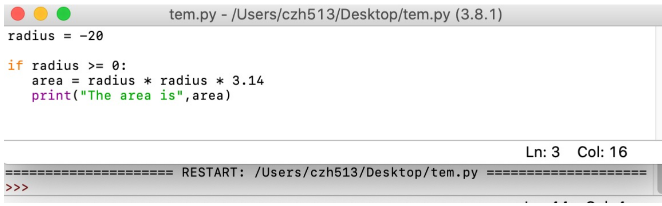
```
radius = -20

if radius >= 0:
    area = radius * radius * 3.14
print("The area is",area)
```

```
>>>
==================== RESTART: /Users/czh513/Desktop/tem.py ====================
Traceback (most recent call last):
  File "/Users/czh513/Desktop/tem.py", line 5, in <module>
    print("The area is",area)
NameError: name 'area' is not defined
>>>
```

(b)

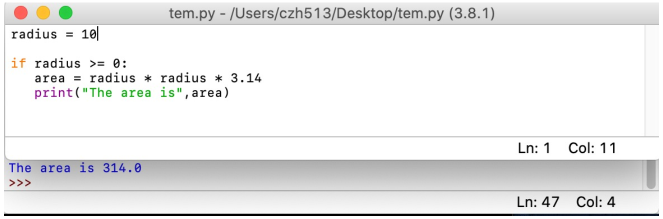
```
radius = -20

if radius >= 0:
    area = radius * radius * 3.14
    print("The area is",area)
```

✓



```
radius = -20

if radius >= 0:
    area = radius * radius * 3.14
    print("The area is",area)
```

```
==================== RESTART: /Users/czh513/Desktop/tem.py ====================
>>>
```

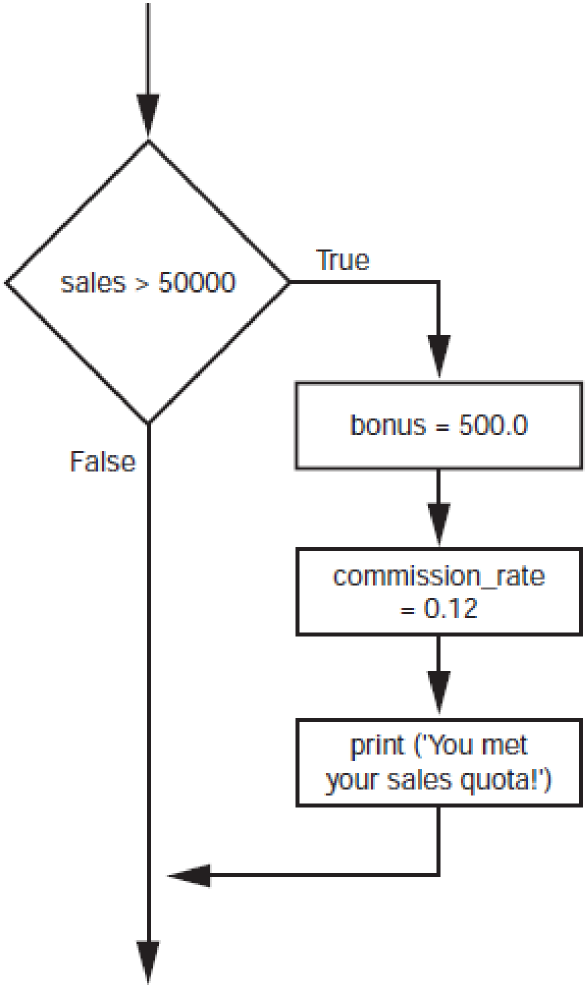


```
radius = 10

if radius >= 0:
    area = radius * radius * 3.14
    print("The area is",area)
```

```
The area is 314.0
>>>
```

Multiple lines of code in an if statement

```
if sales > 50000:
    bonus = 500.0
    commission_rate = 0.12
    print('You met your sales quota!')
```



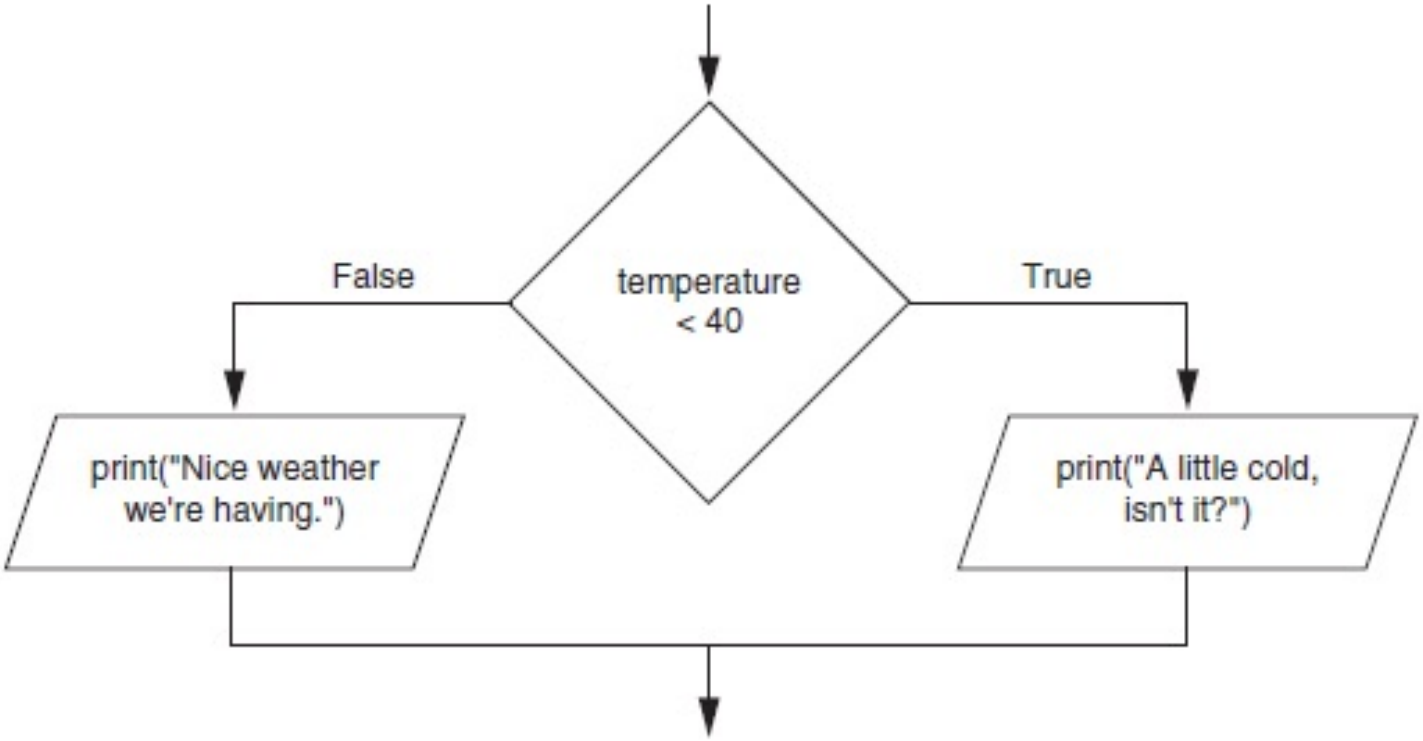
The if-else statement

- **Dual alternative decision structure**: **two** possible paths of execution
  - One is taken if the condition is **`True`**, and the **other** if the condition is **`False`**
  - Syntax:

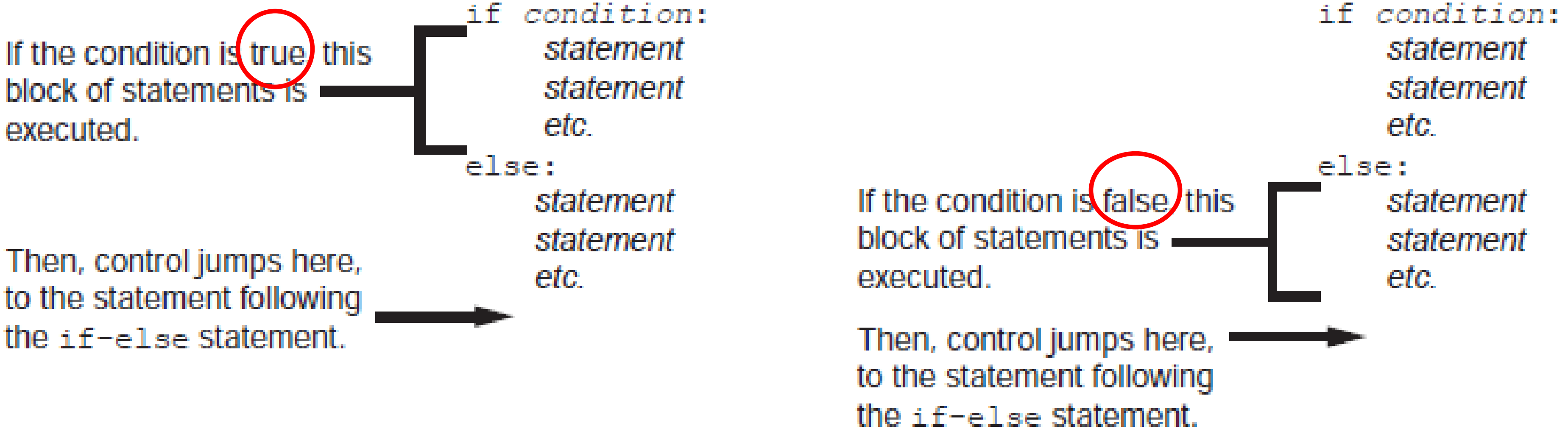
```
if condition:
indent →  statements
else:
indent →  other statements
```

  - **`if`** clause and **`else`** clause **must** be aligned (same indentation)
  - Statements must be **consistently** indented (same indentation)

The if-else statement



The if-else statement



If the condition is true, this block of statements is executed.

```
if condition:
     statement
     statement
     etc.
else:
     statement
     statement
     etc.
```

Then, control jumps here, to the statement following the `if-else` statement.

If the condition is false, this block of statements is executed.

```
if condition:
     statement
     statement
     etc.
else:
     statement
     statement
     etc.
```

Then, control jumps here, to the statement following the `if-else` statement.

Making multiple comparisons using logical operators

- So far, we have just experimented with a single comparison.
- We often need to make multiple comparisons to account for multiple requirements.
- In order to make multiple comparisons we create **multiple conditions** using relational operators and combine them using logical operators.

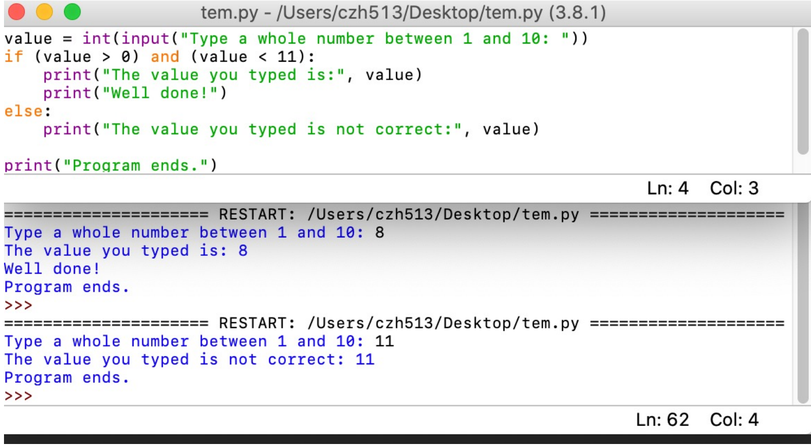
Using the if...else statement

```
value = int(input("Type a whole number between 1 and 10: "))
if (value > 0) and (value < 11):
    print("The value you typed is:", value)
    print("Well done!")
else:
    print("The value you typed is not correct:", value)

print("Program ends.")
```

← multiple conditions

Program Outputs:

```
Type a whole number between 1 and 10: 8
The value you typed is: 8
Well done!
Program ends.

Type a whole number between 1 and 10: 11
The value you typed is not correct: 11
Program ends.
```



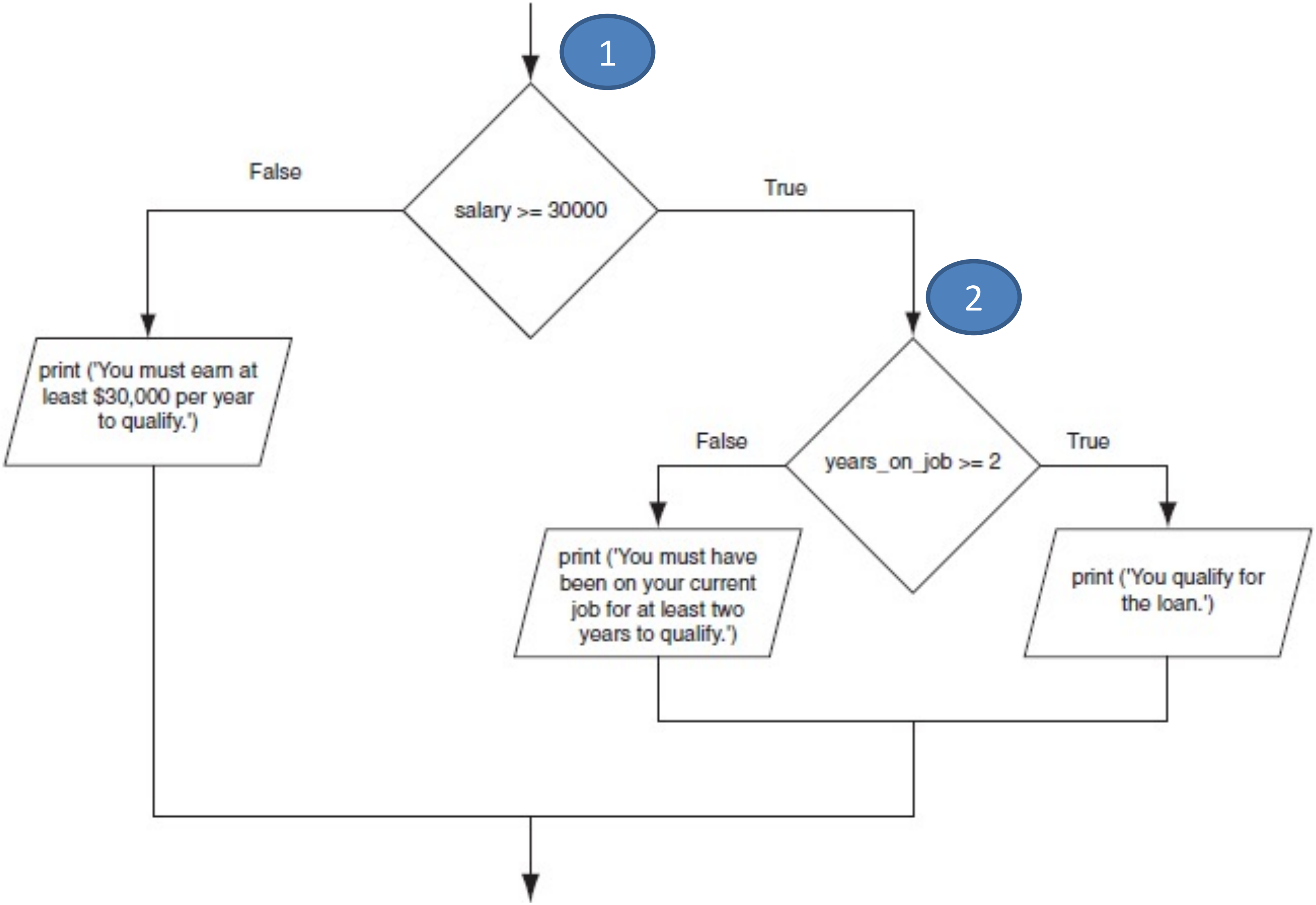
```
value = int(input("Type a whole number between 1 and 10: "))
if (value > 0) and (value < 11):
    print("The value you typed is:", value)
    print("Well done!")
else:
    print("The value you typed is not correct:", value)

print("Program ends.")
```

```
===================== RESTART: /Users/czh513/Desktop/tem.py ====================
Type a whole number between 1 and 10: 8
The value you typed is: 8
Well done!
Program ends.
>>>
===================== RESTART: /Users/czh513/Desktop/tem.py ====================
Type a whole number between 1 and 10: 11
The value you typed is not correct: 11
Program ends.
>>>
```

Nested decision structures

- A decision structure can be nested inside another decision structure (two decisions)
  - This is commonly needed in programs
  - Example:
    - Determine if someone qualifies for a loan, they must meet **two** conditions:
      1 – Must earn at least $30,000/year
      2 – Must have been employed for at least two years
    - Check the first condition, and if it is true, check the second condition

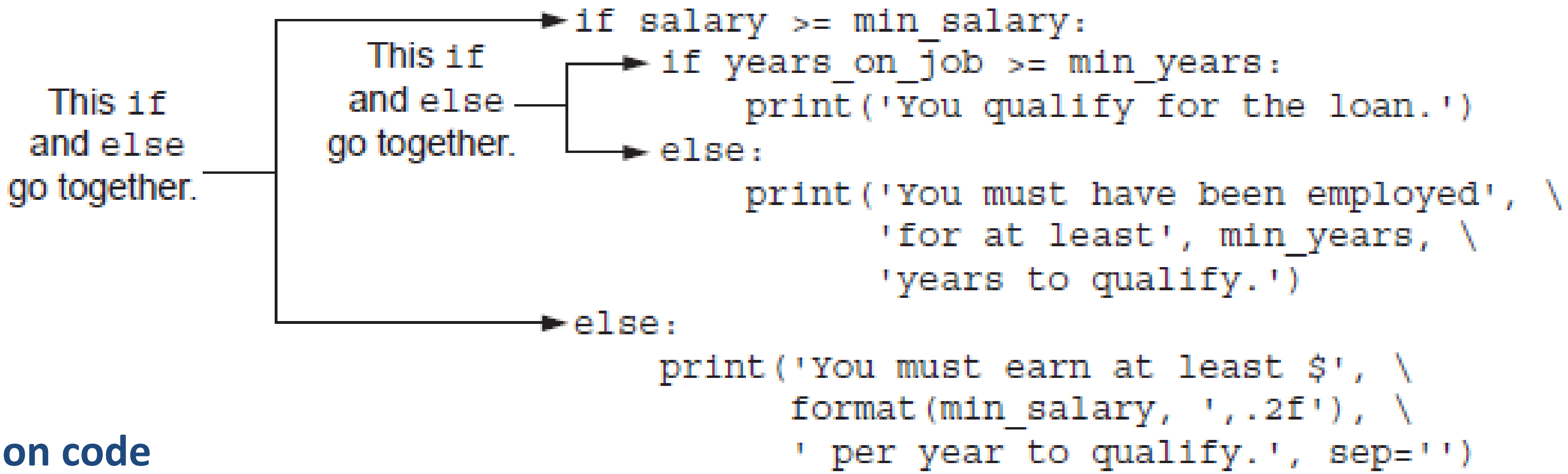
Nested decision structures


1
salary >= 30000
False
True
print ('You must earn at least $30,000 per year to qualify.')
2
years_on_job >= 2
False
True
print ('You must have been on your current job for at least two years to qualify.')
print ('You qualify for the loan.')

Nested decision structures

- It is important to use proper indentation in a nested decision structure
  - Important for Python interpreter as it uses indentation to determine blocks of code
  - Makes code more readable for programmer
  - Reminder: rules for writing nested `if` statements:
    - `else` clause should align with matching `if` clause
    - Statements in each block must be consistently indented

Nested decision structures

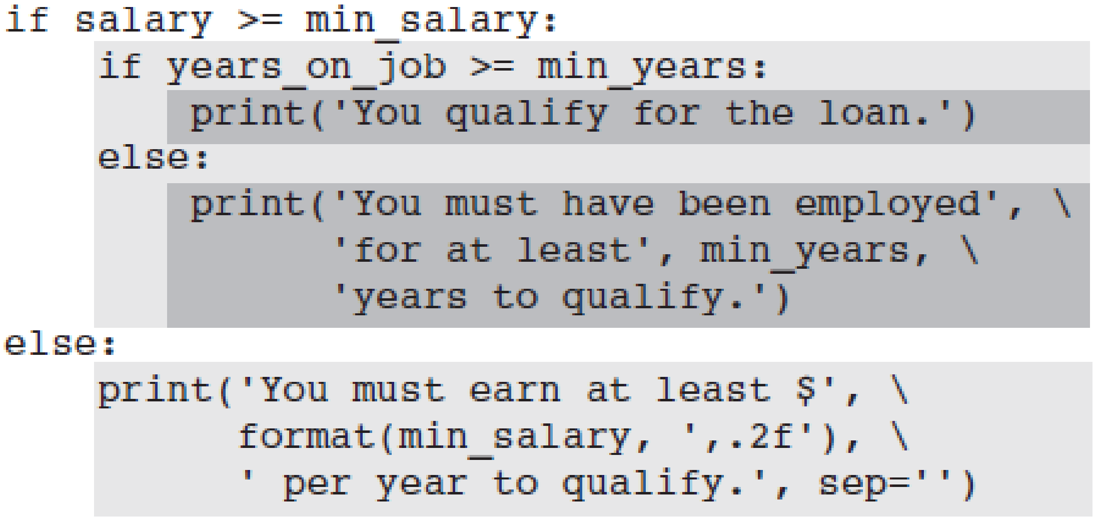


This if and else go together.

This if and else go together.

```
if salary >= min_salary:
    if years_on_job >= min_years:
        print('You qualify for the loan.')
    else:
        print('You must have been employed', \
              'for at least', min_years, \
              'years to qualify.')
else:
    print('You must earn at least $', \
          format(min_salary, ',.2f'), \
          ' per year to qualify.', sep='')
```

Python code



```
if salary >= min_salary:
    if years_on_job >= min_years:
        print('You qualify for the loan.')
    else:
        print('You must have been employed', \
              'for at least', min_years, \
              'years to qualify.')
else:
    print('You must earn at least $', \
          format(min_salary, ',.2f'), \
          ' per year to qualify.', sep='')
```

- A backslash (\) is used to break a long statement into multiple lines to make it more human-readable (the interpreter joins the lines for you)

The if-elif-else statement

- **if-elif-else** statement: a special version of a decision structure. It makes the logic of nested decision structures simpler to write and follow. (Read **elif** as "**else if**")
- Can include multiple **elif** statements and a final (optional) **else**

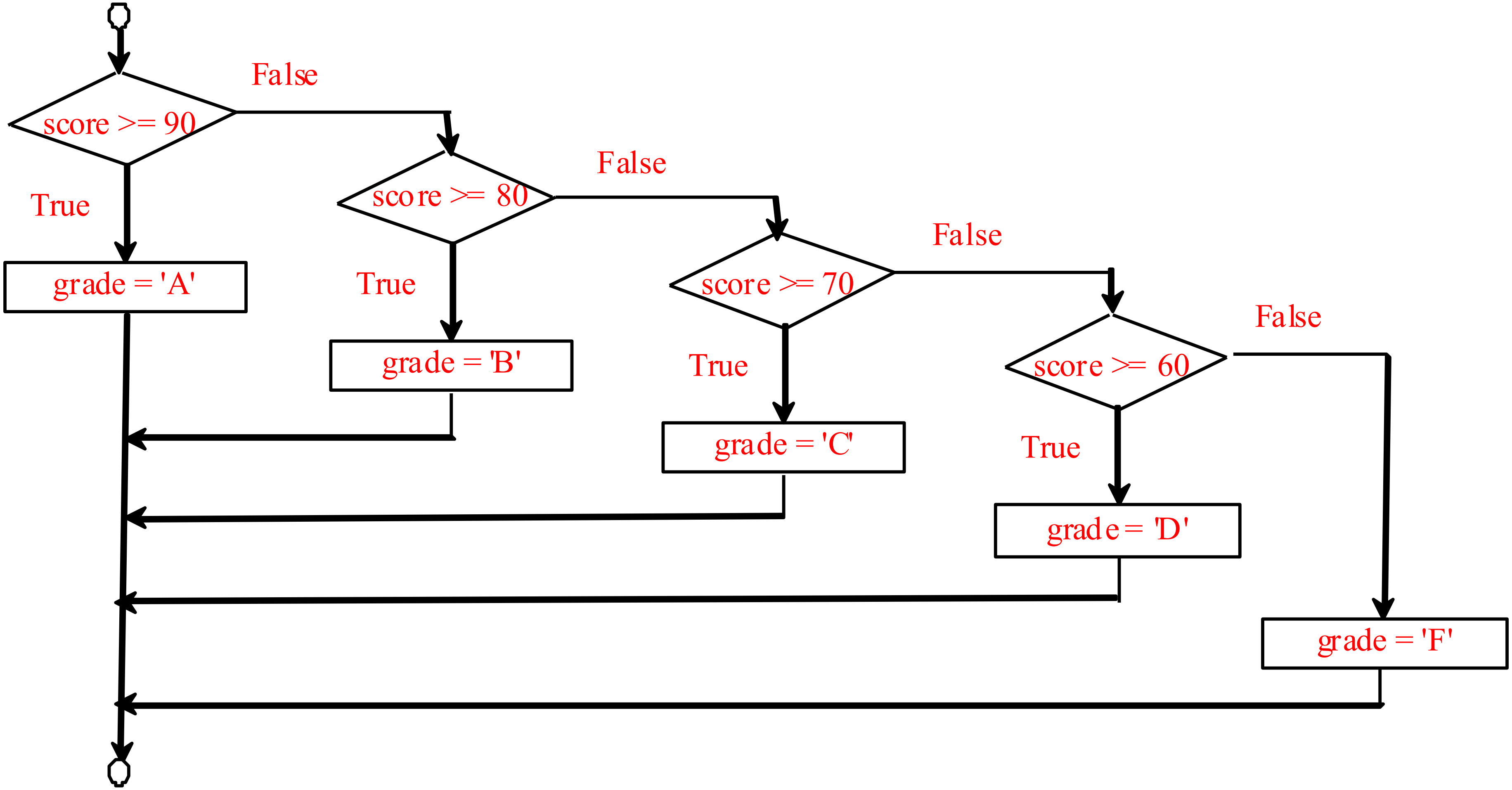
Syntax:

```
if condition 1:
      statements
elif condition 2:
      statements
elif condition 3:
      statements
else:
      statements
```

The if-elif-else statement

- Alignment used with `if–elif–else` statement:
  - `if`, `elif`, and `else` clauses are all aligned the same
  - Conditionally executed blocks are consistently indented
  - the `if–elif–else` statement logic is easier to follow for the programmer than using multiple nested if and else statements
- But can be accomplished by nested `if–else`
  - Code can become complex, and indentation can cause problematic long lines and hard-to-follow logic

Example if flowchart



score >= 90

False

True

grade = 'A'

score >= 80

False

True

grade = 'B'

score >= 70

False

True

grade = 'C'

score >= 60

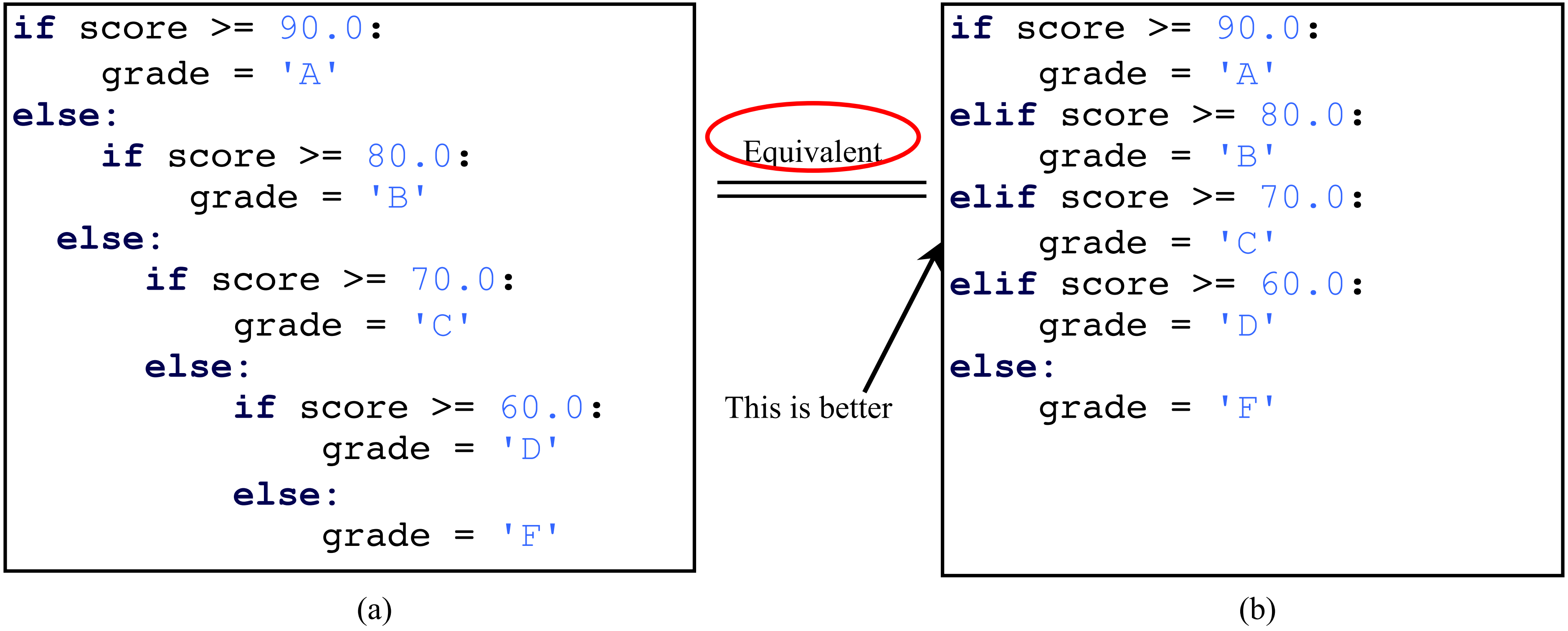
False

True

grade = 'D'

grade = 'F'

Multiple alternative if statements



```
if score >= 90.0:
    grade = 'A'
else:
    if score >= 80.0:
        grade = 'B'
  else:
      if score >= 70.0:
          grade = 'C'
      else:
          if score >= 60.0:
              grade = 'D'
          else:
              grade = 'F'
```

(a)

Equivalent

This is better

```
if score >= 90.0:
    grade = 'A'
elif score >= 80.0:
    grade = 'B'
elif score >= 70.0:
    grade = 'C'
elif score >= 60.0:
    grade = 'D'
else:
    grade = 'F'
```

(b)

Program flow for example if-else statement

Suppose score is 70.0

The condition is false

```
if score >= 90.0:
    grade = 'A'
elif score >= 80.0:
    grade = 'B'
elif score >= 70.0:
    grade = 'C'
elif score >= 60.0:
    grade = 'D'
else:
    grade = 'F'
```

Program flow for example if-else statement

Suppose score is 70.0

The condition is false

```
if score >= 90.0:
    grade = 'A'
elif score >= 80.0:
    grade = 'B'
elif score >= 70.0:
    grade = 'C'
elif score >= 60.0:
    grade = 'D'
else:
    grade = 'F'
```

Program flow for example if-else statement

Suppose score is 70.0

The condition is true

```
if score >= 90.0:
    grade = 'A'
elif score >= 80.0:
    grade = 'B'
elif score >= 70.0:
    grade = 'C'
elif score >= 60.0:
    grade = 'D'
else:
    grade = 'F'
```

Program flow for example if-else statement

Suppose score is 70.0

grade is C

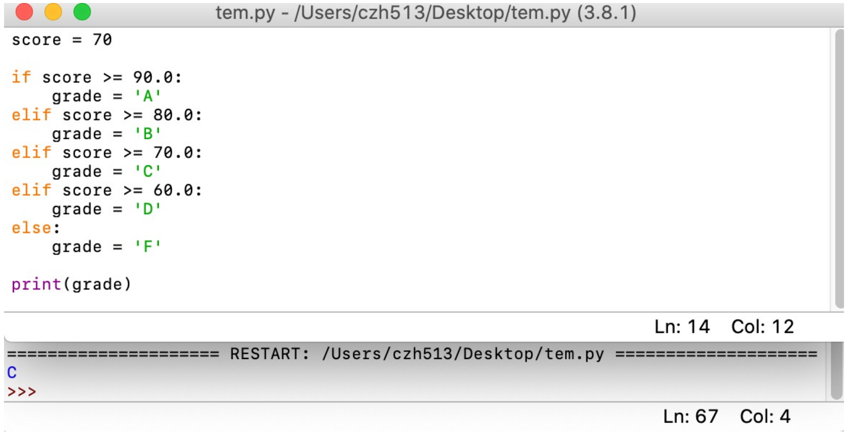
```
if score >= 90.0:
    grade = 'A'
elif score >= 80.0:
    grade = 'B'
elif score >= 70.0:
    grade = 'C'
elif score >= 60.0:
    grade = 'D'
else:
    grade = 'F'
```

Program flow for example if-else statement

Suppose score is 70.0

Exit the if statement

```
if score >= 90.0:
    grade = 'A'
elif score >= 80.0:
    grade = 'B'
elif score >= 70.0:
    grade = 'C'
elif score >= 60.0:
    grade = 'D'
else:
    grade = 'F'
```



```
score = 70

if score >= 90.0:
    grade = 'A'
elif score >= 80.0:
    grade = 'B'
elif score >= 70.0:
    grade = 'C'
elif score >= 60.0:
    grade = 'D'
else:
    grade = 'F'

print(grade)
```

```
==================== RESTART: /Users/czh513/Desktop/tem.py ===================
C
>>>
```

These statements are skipped as we found an elif statement that evaluated to True!

Questions?