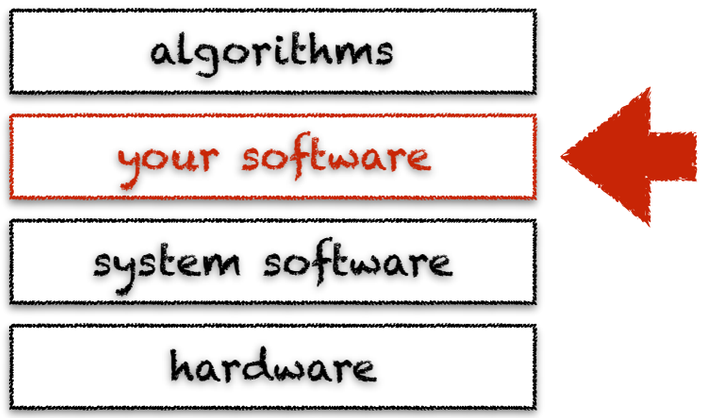
programming basics

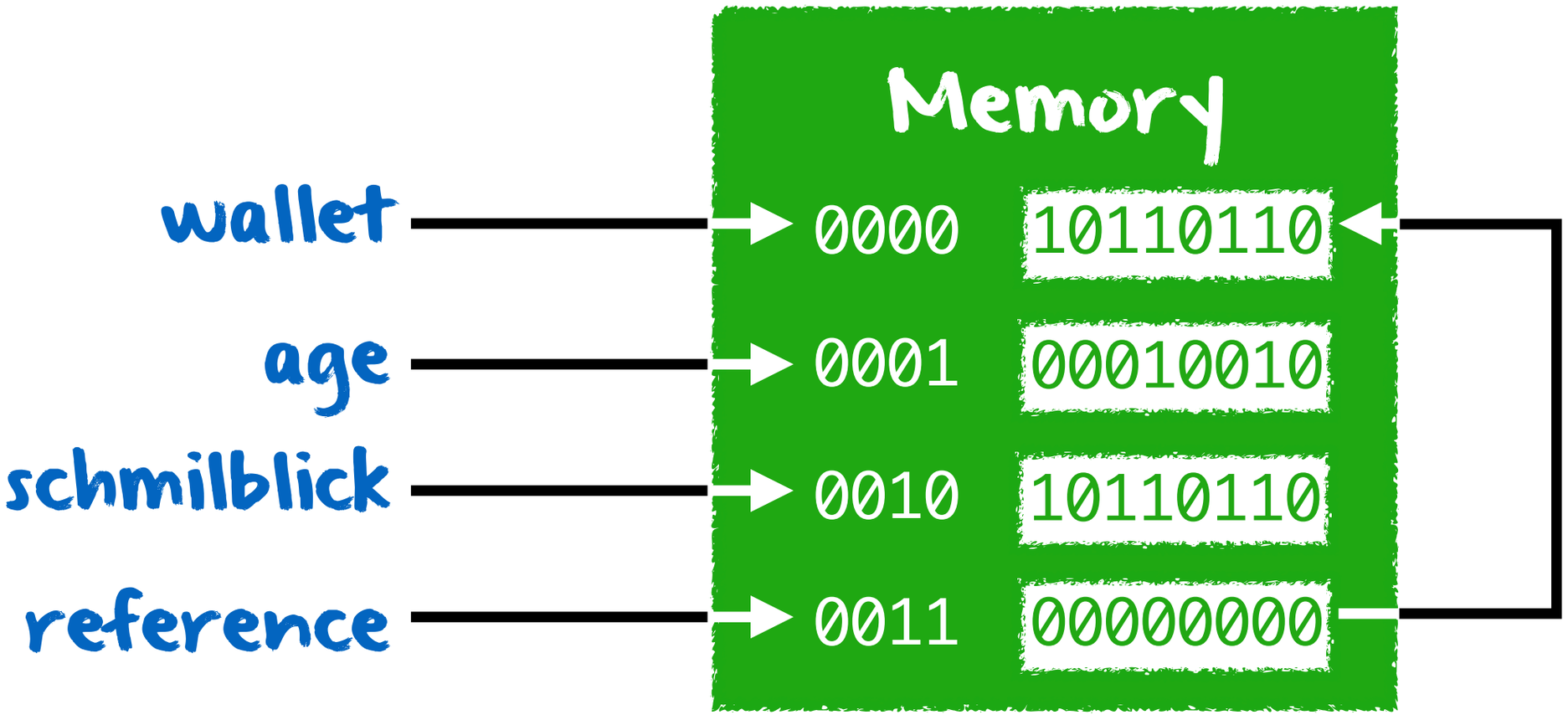
learning objectives



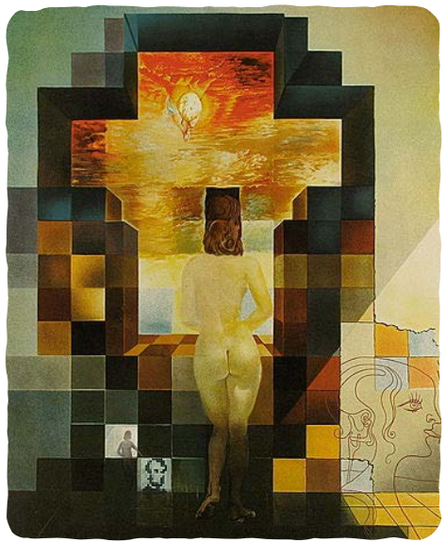
- learn about variables, their types and their values
- learn about different number representations
- learn boolean algebra and conditional branching
- learn about basic text input and output

what's a variable?

in a program, a variable is a symbolic name (also called identifier) associated with a memory location where the value of the variable will be stored



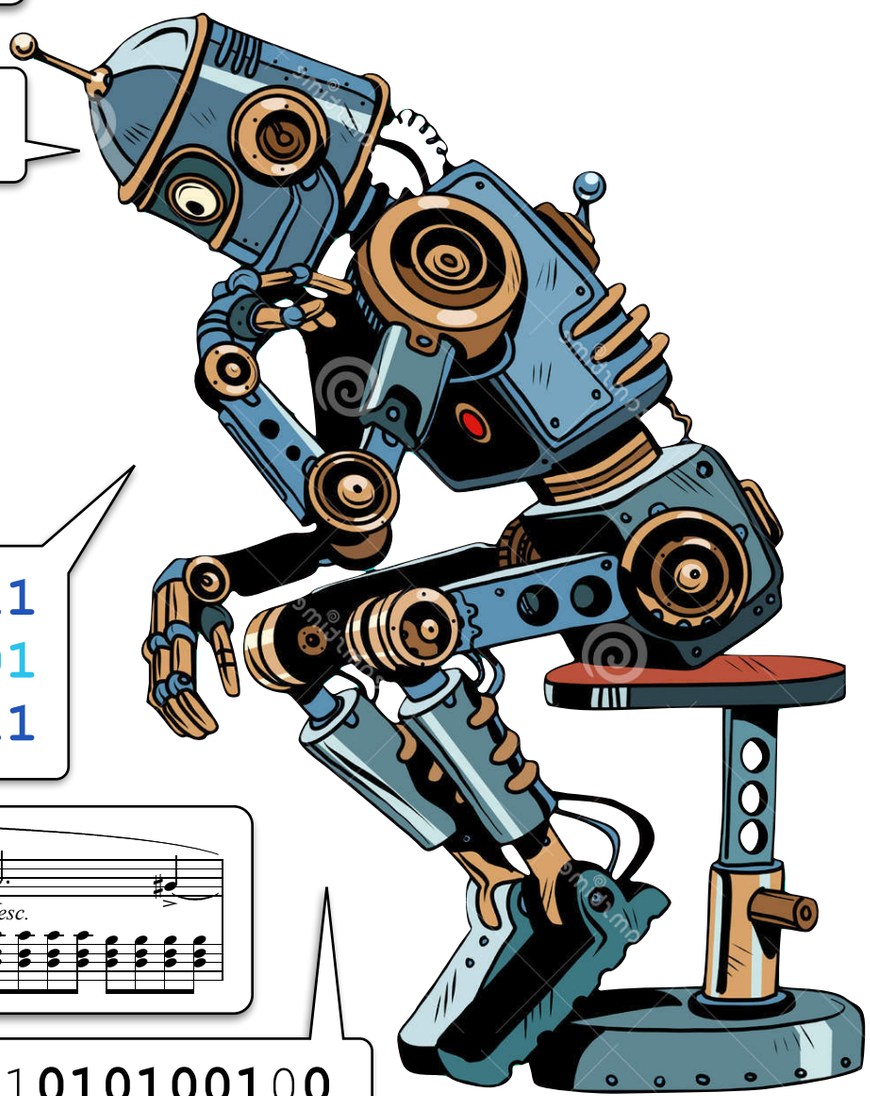
yes but what type of value?



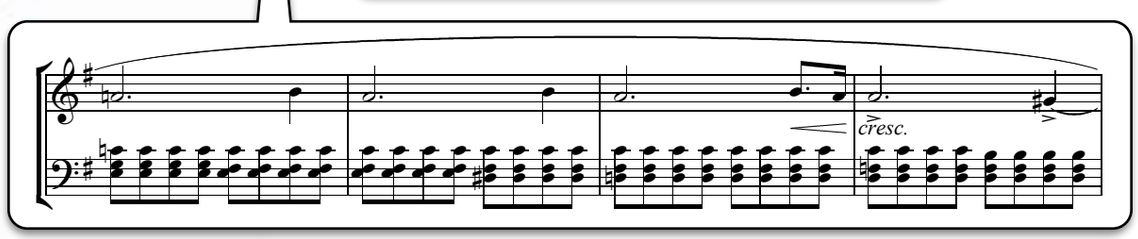
00100101
00101011
00010010
10100100
11001101
00111001
11110011
01010011

$$x^n + y^n = z^n$$

1111001101010011



00100101001011
11001101001110
11110011010011



001001010010101100010010101001000
110011010011100111101101010011

what's a type?

the type of a variable defines what will be stored in the memory location, e.g., a boolean, an integer, a character, etc., i.e., how the bits in the memory location will be interpreted

python	scala	swift

`d = 3.14`
`i = 0`
`s = "hello"`
	`var d = 3.14`
	`var i = 0`
	`var s = "hello"`
		`var d = 3.14`
		`var i = 0`
		`var s = "hello"`

`00111110001000000000000000000000` ⇔ `0.15625`

`1000001` ⇔ `65`

`1000001` ⇔ ‘A’

`0000000` ⇔ false

explicit typing & type inference

as a programmer, you can explicitly define the type of a variable (explicit typing) or let the compiler (or the interpreter) try to infer the type of the variable, typically through initialization (implicit typing)

however, there are cases where type inference is not possible, e.g., in recursive functions

	python	scala	swift

implicit	`i = 0`
	`f = 3.14`
	`s = "hello"`
		`var i = 0`
		`var d = 3.14`
		`var f = 3.14f`
		`var s = "hello"`
			`var i = 0`
			`var d = 3.14`
			`var s = "hello"`

explicit	no static typing	`var i : Int = 0`
		`var f : Double = 3.14`
		`var f : Float = 3.14f`
		`var s : String = "hello"`
			`var i : Int = 0`
			`var f : Double = 3.14`
			`var f : Float = 3.14`
			`var s : String = "hello"`

static typing vs dynamic typing

the static type designates the type of the variable known at compilation time

this allows the compiler to catch a certain number of errors before the execution

scala

```
var i : Int = 0
var d = 3.14
var f = 3.14f
var s = “hello”

f : Float = d
i = d
s = d
```

the dynamic type designates the type of the value contained by a variable at run time

this allows the runtime to catch errors during the execution

python

```
v = 0
v = 3.14
v = “hello”
```

type casting

when you want to assign a value to a variable but the static type and the dynamic type do not match, you can perform an **explicit conversion**, also known as a **type casting**

python	scala	swift

`d = math.pi`
`i = int(d)`
`f = float(d)`
`s = str(d)`
	`var d : Double = math.Pi`
	`var i = 0`
	`var s = "hello"`
	`var f = 3.14f`
	
	`f = d.toFloat`
	`i = d.toInt`
	`s = d.toString`
		`var d : Double.pi`
		`var i = 0`
		`var s = "hello"`
		`var f : Float = 3.14`
		
		`var i = Int(d)`
		`var f = Float(d)`
		`var s = String(d)`

3.141592653589793

3.1415927

3

"3.141592653589793"

number representation

unsigned integers

	bit 7		bit 6		bit 5		bit 4		bit 3		bit 2		bit 1		bit 0
87_{10} =	0×2^7	+	1×2^6	+	0×2^5	+	1×2^4	+	0×2^3	+	1×2^2	+	1×2^1	+	1×2^0
87_{10} =	0×128	+	1×64	+	0×32	+	1×16	+	0×8	+	1×4	+	1×2	+	1×1
87_{10} =	0		1		0		1		0		1		1		1

$87_{10} = 01010111_2$

range = $[0_2,11111111_2] = [0_{10},255_{10}]$

signed integers with signed magnitude

	Bit 7	bit 6		bit 5		bit 4		bit 3		bit 2		bit 1		bit 0
87_{10} =	0	1×2^6	+	0×2^5	+	1×2^4	+	0×2^3	+	1×2^2	+	1×2^1	+	1×2^0
87_{10} =	0	1		0		1		0		1		1		1
-87_{10} =	1	1×64	+	0×32	+	1×16	+	0×8	+	1×4	+	1×2	+	1×1
-87_{10} =	1	1		0		1		0		1		1		1

$87_{10} = 01010111_2$
$-87_{10} = 11010111_2$

Bit 7 is the sign bit

0 ⇔ +
1 ⇔ −

range = $[-127_{10},+127_{10}]$

two ways to represent zero:
$+0_{10} = 00000000_2$
$-0_{10} = 10000000_2$

number representation

signed integers with one complement								
	Bit 7	**bit 6**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**
87_{10} =	0	1×2^6 +	0×2^5 +	1×2^4 +	0×2^3 +	1×2^2 +	1×2^1 +	1×2^0
87_{10} =	0	1	0	1	0	1	1	1
	not ⬇	**not** ⬇	**not** ⬇	**not** ⬇	**not** ⬇	**not** ⬇	**not** ⬇	**not** ⬇
-87_{10} =	1	0	1	0	1	0	0	0

$87_{10} = 01010111_2$
$-87_{10} = 10101000_2$

Bit 7 is the sign bit

0 ⇔ +
1 ⇔ −

range = $[-127_{10},+127_{10}]$

two ways to represent zero:
$+0_{10} = 00000000_2$
$-0_{10} = 11111111_2$

number representation

signed integers with two complement															
	Bit 7	**bit 6**		**bit 5**		**bit 4**		**bit 3**		**bit 2**		**bit 1**		**bit 0**
87_{10} =	0	1×2^6	+	0×2^5	+	1×2^4	+	0×2^3	+	1×2^2	+	1×2^1	+	1×2^0
87_{10} =	0	1		0		1		0		1		1		1
	not ⬇	**not** ⬇		**not** ⬇		**not** ⬇		**not** ⬇		**not** ⬇		**not** ⬇		**not** ⬇
	1	0		1		0		1		0		0		0
														+1 ⬇
-87_{10} =	1	0		1		0		1		0		0		1
	-1×2^7	0×2^6	+	1×2^5	+	0×2^4	+	1×2^3	+	0×2^2	+	0×2^1	+	1×2^0
-87_{10} =	-1×128		+	1×32			+	1×8					+	1×1

$87_{10} = 01010111_2$
$-87_{10} = 10101001_2$

Bit 7 is the sign bit

0 ⇔ +
1 ⇔ −

range = $[-128_{10},+127_{10}]$

only one way to represent zero:
$0_{10} = 00000000_2$

number representation

only a small subset of the infinite set of real numbers can be represented in a computer, which has a finite memory space

floating point principle

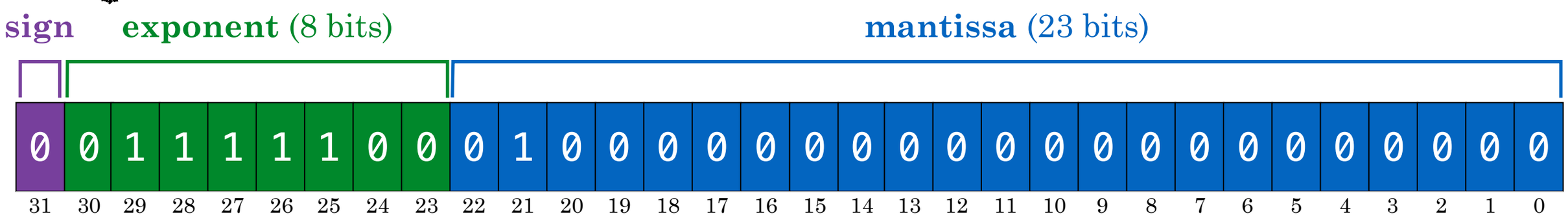
$$\text{sign} \times \text{mantissa} \times \text{base}^{\text{exponent}}$$

$$-3.14159 = -1 \times 314159 \times 10^{-5}$$

in a computer, the base is 2

number representation

floating point single precision



$$\text{value} = (-1)^{\text{sign}} \times \left(1 + \sum_{i=1}^{23} b_{23-i}\, 2^{-i}\right) \times 2^{(e-127)}$$

$$\text{sign} = b_{31} = 0 \implies (-1)^{\text{sign}} = (-1)^0 = +1 \in \{-1, +1\}$$

$$e = b_{30}b_{29}\ldots b_{23} = \sum_{i=0}^{7} b_{23+i}2^{+i} = 124 \in \{1, \ldots, (2^8 - 1) - 1\} = \{1, \ldots, 254\}$$

$$2^{(e-127)} = 2^{124-127} = 2^{-3} \in \{2^{-126}, \ldots, 2^{127}\}$$

$$1.b_{22}b_{21}\ldots b_0 = 1 + \sum_{i=1}^{23} b_{23-i}2^{-i} = 1 + 1 \cdot 2^{-2} = 1.25 \in \{1, 1 + 2^{-23}, \ldots, 2 - 2^{-23}\} \subset [1; 2 - 2^{-23}] \subset [1; 2)$$

$$\text{value} = (+1) \times 1.25 \times 2^{-3} = +0.15625$$

constant

a constant is simply a variable that cannot... vary 😜

python	scala	swift

no constant	`val d : Double = math.Pi`
	`val i = 0`
	`val s = "hello"`
	
	`d = 1.0`
	`i = 1`
	`s = "bye"` 
		`let d : Double.pi`
		`let i = 0`
		`let s = "hello"`
		
		`d = 1.0`
		`i = 1`
		`s = "bye"` 

logic

the intellectual tool for reasoning about the truth and falsity of statements

logic & programming

most programming languages, support **boolean variables**, which can take values $\in \{\texttt{true}, \texttt{false}\}$

in some low-level languages, integer numbers are used for the same purpose, e.g,. with:

```
p = false   ⇔  p = 0
q = true    ⇔  q = 1 (sometimes  q = true  ⇔  q ≠ 0 )
```

when combined with operators $\wedge$, $\vee$ and $\neg$, boolean variables constitute an algebra used in **conditional branching**

where: $\neg \Leftrightarrow$ not
$\vee \Leftrightarrow$ or
$\wedge \Leftrightarrow$ and

boolean algebra

assume that p , q and r are boolean variables (or statements) and that $T = true$, $F = false$, we have:

p	$\neg p$
F	T
T	F

p	q	$p \wedge q$
F	F	F
F	T	F
T	F	F
T	T	T

p	q	$p \vee q$
F	F	F
F	T	T
T	F	T
T	T	T

$\neg \Leftrightarrow$ **not**
$\vee \Leftrightarrow$ **or**
$\wedge \Leftrightarrow$ **and**

python	scala	swift

`a = False`
`b = True`
 	
`c = a and b`
`c = a or b`
`c = not a`
	`var a = false`
	`var b = true`
	
	`var c = a && b`
	`c = a || b`
	`c = !a`
		`var a = false`
		`var b = true`
		
		`var c = a && b`
		`c = a || b`
		`c = !a`

some rules

Associative Rules:	$(p \wedge q) \wedge r \Leftrightarrow p \wedge (q \wedge r)$	$(p \vee q) \vee r \Leftrightarrow p \vee (q \vee r)$
Distributive Rules:	$p \wedge (q \vee r) \Leftrightarrow (p \wedge q) \vee (p \wedge r)$	$p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$
Idempotent Rules:	$p \wedge p \Leftrightarrow p$	$p \vee p \Leftrightarrow p$
Double Negation:	$\neg\neg p \Leftrightarrow p$	
DeMorgan's Rules:	$\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$	$\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$
Commutative Rules:	$p \wedge q \Leftrightarrow q \wedge p$	$p \vee q \Leftrightarrow q \vee p$
Absorption Rules:	$p \vee (p \wedge q) \Leftrightarrow p$	$p \wedge (p \vee q) \Leftrightarrow p$
Bound Rules:	$p \wedge F \Leftrightarrow F \quad p \wedge T \Leftrightarrow p$	$p \vee T \Leftrightarrow T \quad p \vee F \Leftrightarrow p$
Negation Rules:	$p \wedge (\neg p) \Leftrightarrow F$	$p \vee (\neg p) \Leftrightarrow T$

from boolean algebra to conditional branching

example

write a function that checks whether a given year (passed as parameter) is a leap year or not

Leap years are multiples of 4, and they can only be multiples of 100 if they are also multiples of 400

```
function isLeap(year : integer)
if year mod 400 = 0
    isLeap ← true
else if year mod 100 = 0
    isLeap ← false
else if year mod 4 = 0
    isLeap ← true
else isLeap ← false
```

conditional branching

```
function isLeap(year : integer)
if ((year mod 4 = 0) ∧ (year mod 100 ≠ 0)) ∨ (year mod 400)
    isLeap ← true
else
    isLeap ← false
```

```
function isLeap(year : integer)
isLeapYear ← ((year mod 4 = 0) ∧ (year mod 100 ≠ 0)) ∨ (year mod 400)
```

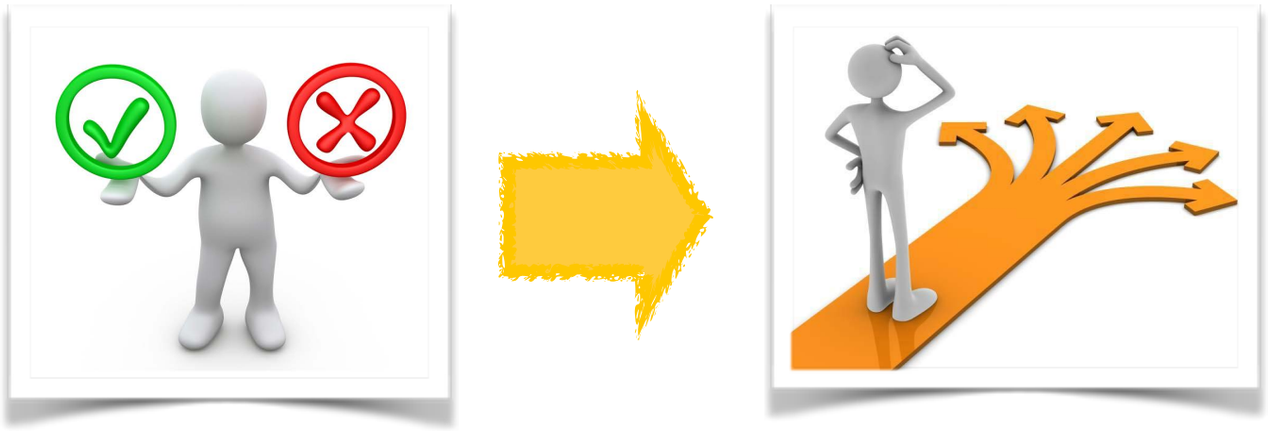
conditional branching

python

```
def isLeap(year):
    if year % 400 == 0 : return True
    elif year % 100 == 0 : return False
    elif year % 4 == 0 : return True
    return False
```

```
def isLeap(year):
    if (year % 4 == 0) and (year % 100 != 0) or (year % 400 == 0) : return True
    return False
```

```
def isLeap(year):
    return (year % 4 == 0) and (year % 100 != 0) or (year % 400 == 0)
```



conditional branching

scala

```
def isLeap(year : Int) : Boolean = {
  if  (year % 400 == 0) true
  else if (year % 100 == 0) false
  else if (year % 4 == 0) true
  else false
}
```

```
def isLeap(year : Int) : Boolean = {
  if  ((year % 4 == 0) && (year % 100 != 0) || (year % 400 == 0)) true
  else false
}
```

```
def isLeap(year : Int) : Boolean =
(year % 4 == 0) && (year % 100 != 0) || (year % 400 == 0)
```

conditional branching

swift

```
func isLeap(year:Int) -> Bool {
     if year % 400 == 0 { return true }
     else if year % 100 == 0 { return false }
     else if year % 4 == 0 { return true }
     else  { return false }
}
```

```
func isLeap(year:Int) -> Bool {
     if (year % 4 == 0) && (year % 100 != 0) || (year % 400 == 0) { return true }
     else  { return false }
}
```

```
func isLeap(year:Int) -> Bool {
     return (year % 4 == 0) && (year % 100 != 0) || (year % 400 == 0)
}
```

scala

```
i match {
  case 1  => println("January")
  case 2  => println("February")
  case 3  => println("March")
  case 4  => println("April")
  case 5  => println("May")
  case 6  => println("June")
  case 7  => println("July")
  case 8  => println("August")
  case 9  => println("September")
  case 10 => println("October")
  case 11 => println("November")
  case 12 => println("December")
  case whoa  => println("Unexpected: " + whoa.toString)
}
```

conditional branching

switch / match

swift

```
let someCharacter: Character = "z"
switch someCharacter {
case "a":
  print("The first letter of the alphabet")
case "z":
  print("The last letter of the alphabet")
default:
  print("Some other character")
}
```

fallback case

reserved keywords

in a programming language, identifiers are lexical tokens chosen by the programmer to name various kinds of entities, e.g., variables, functions, types, etc.

in contrast, reserved keywords are words that cannot be chosen by the programmer to name entities and that has a predefined meaning, if, else, switch, etc.

command line arguments

scala

```scala
object HelloWorld extends App {
    if (args.length == 0) {
        println("Hello world")
    } else {
        println("Hello " + args(0))
    }
}
```

```
Args-Scala — -bash — ttys000
[wallace-palace:Args-Scala garbi$ scalac HelloWorld.scala
[wallace-palace:Args-Scala garbi$ scala HelloWorld
Hello world
[wallace-palace:Args-Scala garbi$ scala HelloWorld Donald
Hello Donald
wallace-palace:Args-Scala garbi$
```

text input/output on the command line

when a program is launched on the command line, it can ask the user for text input and provide text output on the terminal

	input	output
python	`year = input("Give us a year: “)` `year = int(year)`	`print("Is {0} a leap year? {1}".format(year, isLeap(year)))`

scala	`import scala.io.StdIn.readLine`
	
	`val year = readLine("Choose a year: ").toInt`
		`print(s"Is $year a leap year? ${isLeap(year)}")`

swift	`var year = Int(readLine()!)`	`print("Year \(year!) is leap: \(isLeap(year:year!))")`